Логическо програмиране

Антон Зиновиев

10 декември 2025 г.

Съдържание

Глава 1

Математическа логика и програмиране

1.1. Безсмисленото начало

> *Каква е ползата от Вашите красиви изследвания за π? Защо да се занимаваме с такива неща при положение, че ирационалните числа не съществуват?*
>
> Леополд Кронекер

В математиката винаги явно или неявно са се използвали множества, но в продължение на дълго време множествата били образувани по сравнително прост начин. Имало е множества от реални числа, множества от функции между реални числа, множества от точки и т. н. и винаги е имало ясно и разбираемо правило, определящо кои точно са елементите на дадено множество. Математиците не се увличали по измамни занимания с далечни от ума обекти като множества, които съдържат всички подмножества на множеството, което съдържа всички подмножества на множеството, което съдържа всички подмножества на множеството, което съдържа всички естествени числа.

През последната четвърт на 19-ти век обаче се появила съблазън, която разклатила здравомислието на математиците. Основна заслуга за това имал Георг Кантор, който неблагоразумно дръзнал да разработи основите на това, което днес се нарича *теория на множествата*. За

съжаление, оказало се, че заниманията с теория на множествата неизбежно изискват от математѝка да описва свойствата на обекти, които поради самия им характер е невъзможно човек да си ги представи ясно. Тогава малцина съзрели колко опасни могат да бъдат тези прелъстителни занимания, а с особена прозорливост сред тях се отличил Леополд Кронекер. Той не спирал доблестно да предупреждава всички, че Кантор е псевдоучен, шарлатанин, ренегат и развратител на младежта, но — уви — малцина се вслушали в гласа на разума. И наказанието, не се забавило...

Едно след друго в теория на множествата се появили противоречия. Така например Кантор доказал, че за всяко множество X множеството 2^X има повече елементи от X. Но какво правим когато X е множеството, което съдържа всички множества? Как така 2^X ще съдържа повече елементи от множеството, което съдържа всички множества? За да избегне това противоречие, Кантор решил, че множеството, което съдържа всички множества, е твърде голямо и затова не можело да бъде множество. Друг парадокс е свързан с т. н. ординални числа. Оказало се, че от всяко ординално число има по-голямо, а в същото време множеството от всички ординални числа трябва да бъде най-голямото ординално число. И в този случай Кантор решил, че множеството от всички ординални числа е твърде голямо и затова не можело да бъде множество. Друго известно противоречие е парадоксът на Ръсел.[*] За да бъде преодоляно то, през 1908 Цермело измислил аксиоми, според които множествата се строят на безкрайна трансфинитна редица от стъпки като всяко множество може да съдържа само елементи, които вече са били образувани на някоя предшестваща стъпка. По този начин математиката започнала да прилича на съвременна компютърна програма, която е пълна с грешки и която след всяка новооткрита грешка трябва да се оправя.[**]

Кронекер починал, но призивът за конструктивизъм в математиката бил подет от други математици, напр. Анри Поанкаре. Особено активен бил Брауер, който измислил нов вид конструктивна математика, която нарекъл *интуиционизъм*.[***] Брауер установил, че ако искаме

[*] Обичайните множества не съдържат себе си като елементи. Да наречем такива множества *нормални* и нека R е множеството, което съдържа всички нормални множества. Самото R нормално множество ли е?

[**] Все още не е открито противоречие в аксиомите на Цермело.

[***] Името „интуиционизъм“ идва от това, че според Брауер математиката трябва да включва само конструкции, които можем да си ги представим ясно. Благодарение на това математикът може да установява правилността на една конструкция посредством просто позоваване на интуицията си.

да разработваме математиката конструктивно, е удобно да използваме логика, която е различна от обичайната. Така например в интуиционизма не е вярно, че всяко съждение е вярно или невярно, но въпреки това няма съждение, което едновременно не е вярно и не е невярно.

С огорчение Давид Хилберт гледал как дори собствените му ученици започвали да се увличат от идеите на интуиционизма. А когато към интуиционистите се присъединил и Херман Вейл, Хилберт решил да се намеси. Той заклеймил интуиционистите като метежници, организирали идеологическа чистка, които без да разполагат с доказателство за виновност унищожавали всичко, за което решат, че буди подозрения. А конкретно за Брауер той писал: „Аз съм безкрайно удивен от факта, че дори и сред математиците се оказва, че силата на внушението на един единствен човек, без значение колко темпераментен или остроумен, е в състояние да предизвика толкова невероятни и чудати въздействия.“

През 1922 г. Хилберт обявил своята знаменита програма за „спасяване“ на математиката. Тя се състои в следното:

(1). Дефиниране на формално точен математически език, на който може да се формулират математическите съждения. Така формулираните математически съждения се наричат *формули*.

(2). Формулиране на точно определени правила, посредством които от вече доказани формули можем да получаваме нови доказани формули.

(3). Доказателство, че така формулираните правила са достатъчни, за да се докажат всички математически верни съждения.

(4). Доказателство, че с така формулираните правила не може да се стига до противоречие. В това доказателство Хилберт иска да се използват само методи, които биха били убедителни дори и за Брауер.

(5). Доказателство, че когато използваме в доказателствата сложни и неконструктивни обекти, ще можем да правим по-кратки доказателства, но няма да може да се докажат неща, които не могат да се докажат и без използването на неконструктивни обекти.

(6). Намиране на метод, посредством който може да се решава дали едно математическо съждение е вярно.

Отначало нещата потръгнали добре. Още през 1879 г. Готлоб Фреге бил дефинирал логически език, който е еквивалентен по изразителна сила на съвременната предикатна логика от първи ред и на който може да се формулира цялата математика. Освен това Фреге формулирал и точни правила, посредством които от вече доказани формули може да

се получават нови доказани правила. С това първата и втората точка от плана на Хилберт можело да се считат за изпълнени.

Някои значителни успехи били постигнати и по отношение на останалите точки от плана. Например през 1929 г. Курт Гьодел доказал, че правилата на Фреге са достатъчни, за да се докаже с тях всяка логическа истина. Много на брой и извънредно важни резултати били получени в продължение само на няколко години. Това било окуражаващо и вдъхновяващо и изглеждало, че аха-аха, още съвсем малко и програмата на Хилберт ще бъде изпълнена.

Но през 1931 г. се случило немислимото — Гьодел доказал своите знаменити теореми за непълнота. От тях следвало, че:

- ако една математическа теория включва в себе си аритметиката,[*] то е невъзможно с краен брой формални правила и аксиоми да се докажат всички верни неща в нея;
- ако една математическа теория включва в себе си аритметиката, то тогава нейната непротиворечивост не може да се докаже, използвайки само методи, които се включват в тази теория.

Това означава, че по колкото и хитър начин да формулираме аксиомите, винаги ще има верни математически съждения, които не следват от тях. И също така означава, че ако решим да се ограничим само с онези съждения, които следват от аксиомите, няма да може да докажем, че от избраните аксиоми не може да се получи противоречие. Малко по-късно (през 1936 г.) Чърч и Тюринг показали, че и последната точка от плана на Хилберт е неосъществима — няма алгоритмичен метод, посредством който можем да познаваме дали едно математическо съждение е вярно или не. Оригиналният план на Хилберт се оказал неосъществим...[**]

В резултат от усилията да се изпълни програмата на Хилберт се появил един нов раздел в математиката — *математическата логика*. Този нов раздел обаче не успял да осъществи това, заради което бил създаден. Той не успял да подсигури основите на математиката, а обикновените математици продължили да си правят математика без

[*] По точно, ако в нея може да се дефинира какво значи събиране и умножение на естествени числа.

[**] Всъщност през 1936 г. Герхард Генцен успял да докаже по задоволителен начин непротиворечивостта на аритметиката. Няколко десетилетия по-късно Гаиши Такеучи доказал непротиворечивостта на различни теории, които са достатъчно богати, за да обхванат по-голямата част от съвременната математика. Засега обаче не знаем дори как да подходим към проблема за непротиворечивостта на теорията на множествата на Цермело, а тя все още се счита за основа на съвременната математика.

много-много да се интересуват какво се случва в математическата логика. Изобщо математическата логика била абстрактна математическа дисциплина, която се оказала толкова безполезна за каквото и да е, че според Мартин Давис, когато той бил студент (1944 – 1950 г.), дори тополозите* се присмивали на логиците, че витаели някъде далеч в космическото пространство.

Но това съвсем скоро щяло да се промени...

1.2. Денят не си личи по заранта

Самата конструкция е изкуството,
а приложението ѝ в света — зъл паразит.

Лойтцен Егбертъс Ян Брауер

Въпреки че приложенията на математическата логика в останалите дялове на математиката все още са като цяло незначителни,** приложенията ѝ в областта на информатиката и компютрите са толкова много и нерядко неочаквани, че човек неволно си задава въпроса защо това е така. Наистина, и други дялове на математиката имат компютърни приложения, напр. линейната алгебра, геометрията, теория на числата, числените методи, теория на вероятностите, теория на графите и комбинаториката. На нито един от тях обаче приложенията не са толкова разнообразни и всеобхватни, колкото приложенията на математическата логика.

Традиционно математическата логика се разделя на четири поддяла. Всеки един от тях има важни „компютърни“ приложения. Тези поддялове са:

– обща логика и теория на моделите;

*Топологията е полезна математическа дисциплина, в която заедно с полезните неща има и има страшно много теореми, които вероятно никога няма да получат каквото и да е практическо приложение.

**Разбира се те са незначителни само на фона на големите очаквания. Ето някои немаловажни математически приложения на математическата логика. Теория на моделите има някои приложения в алгебрата, например доказателството на основната теорема на алгебрата (това, че всяко поле има алгебрически затворено разширение) се опростява, ако се използва логическата теорема за компактност. В математическия анализ обосноваването на инфинитезималното смятане на Лайбниц с безкрайно малки величини може да се счита за едно от най-важните постижения на математиката на 20-ти век. Много от резултатите в топологията пък са тясно свързани с дълбоки резултати в теория на множествата, а основните приложения на т. н. *дескриптивна теория на множествата* са във функционалния анализ, ергодичната теория и операторната алгебра.

– теория на множествата;
– теория на изчислимостта;
– теория на доказателствата и конструктивна математика.

Обща логика и теория на моделите

Машините, които хората ползвали от праисторическо време, променяли свойствата на различни материални обекти. Например пещта на грънчаря преобразувала неопечените глинени съдове в годна за използване керамика, каруцата на търговеца променяла географските координати на натоварените материални ценности, така че стойността им на новите координати да бъде по-голяма и т. н. Устройствата, които работели с нематериални обекти, т.е. данни, били малко на брой и сравнително прости.

През 1642 г. Блез Паскал изобретил първата механична сметачна машина. Сметачните машини били първите машини, които можели да преобразуват данни, но както при всички останали машини от онова време, действията, които те извършвали, се контролирали непосредствено от човек.

През 1805 г. Жозеф Мари Жакард изобретил тъкачен стан, който можел да тъче платове с повтарящи се орнаменти. Машината можела да чете данните, описващи нужните орнаменти, от перфокарти. Това била първата машина, чиито операции не се управлявали непосредствено от човек, а зависели от програма, записана в паметта на машината.

След като тези две изобретения били комбинирани, се получили т. н. *програмируеми сметачни машини* — сметачни машини, които се управляват не непосредствено от човек, а от програма. Първият проект за програмируема сметачна машина била „Аналитичната машина“ на Чарлз Бебидж (1837 г.). Програми за тази машина са писани от Ада Лавлейс, поради което тя днес се счита за пръв програмист.

За съжаление работата на Бебидж останала малко известна и не е използвана от по-късните конструктори. Първата действително конструирана и работеща програмируема сметачна машина е „Колумбийският диференциален табулатор“, създаден почти един век по-късно (1930 г.) от фирмата Ай Би Ем. Вестниците, винаги радващи се на сензации, нарекли тази машина „суперкомпютърно устройство с умствените способности на 100 математика, дори при решаване на много сложни алгебрични задачи“.

Възможностите на програмируемите сметачни машини бързо се увеличавали и машината „Цет 3“, създадена от Конрад Цузе през 1941 г.

била първото устройство, което като изключим по-малката му памет и скорост на работа, има на теория същите изчислителни възможности както и съвременните компютри.

За да получим *компютър*, оставало да се реализира само още един ключов елемент — за програмируемите сметачни машини програмите и обработваните от тях данни били две напълно различни неща. Те можели да четат програмите си, можели да четат и входните си данни, но можели да записват само данни. Това ограничение означава, че за тях не може да се пишат компилатори, защото генерираният от компилатора изпълним код представлява хем данни, хем програма. Първият компютър, при който не се прави разлика между данни и програми, е машината „Ес Ес И Си“ на Ай Би Ем (1948 г.), вторият — машината „EDSAC“ на Кеймбриджкия университет (1949 г.) и третият — машината „МЕСМ“ на Киевския институт по електротехнология (1950 г.).

Това, че в данните, подавани на един компютър, се съдържат и инструкциите за компютъра, означава, че имаме нужда от формален език, на който да се формулират тези инструкции. Математическата логика е разделът от математиката, който ни дава техники за дефиниране на синтаксиса и семантиката на различни формални езици и изследване на свойствата на тези езици. Формалните езици, които могат да бъдат полезни на практика, далеч не се ограничават само с различни видове езици за програмиране. Всеки формален език, който допуска ефективна обработка, обикновено се оказва или полезен, или много полезен.

* * *

Много и най-разнообразни езици са дефинирани в математическата логика, но ако трябва само един от тях да бъде наречен „Езикът на математическата логика“, то честта я има *предикатната логика от първи ред*. Този език притежава голяма изразителна сила — почти цялата съвременна математика може да бъде формулирана на него. Освен това той е и много удобен за използване — нерядко в математически статии твърденията се формулират, използвайки този език, а не на естествен език (български, английски и т. н.). А приложението на този език като език за заявки към бази данни представлява един от най-впечатляващите примери за компютърно приложение на математическата логика.

По принцип, една от пречките за ефективна обработка на предикатни формули е използването в тях на т. н. свързани променливи. Тази пречка била преодоляна още през 1948 г. от Алфред Тарски, който в желанието си да алгебризира предикатната логика, измислил *цилиндричните алгебри*. Езикът на цилиндричните алгебри има същата из-

разителна сила като езика на предикатната логика с равенство,* но за разлика от него не притежава свързани променливи.

Оставало само да се види как идеите от цилиндричните алгебри можели да се реализират ефективно. Това направил Едгар Код през 1969 г., предлагайки *релационния модел за управление бази данни*. Код доказал, че релационният модел притежава същата изразителна сила, както и предикатната логика от първи ред. Освен това този модел допуска и изключително ефективна реализация — ако разполагахме с възможност за неограничена паралелизация, тогава заявките към една релационна база данни биха се изпълнявали за константно време без значение какъв е нейният размер. Съвсем скоро след публикацията на Код се появил и езикът ес кю ел, който реализира релационния модел и е най-популярен език за управление на бази данни вече повече от 40 години.**

* * *

Един друг важен за математическата логика език е езикът на λ-*смятането* измислено от Аланзо Чърч (1930 – 1936 г.). Също както предикатната логика е дала математическата основа на релационните бази данни, така λ-смятането е дало математическата основа на функционалното програмиране. И също както в предикатното смятане има свързани променливи, поради които то е неудобно за непосредствена реализация, така и в λ-смятането има свързани променливи. Само че докато при базите данни хората (поне засега) се мъчат и вместо езици със свързани променливи използват по-неудобни езици като ес кю ел, то при функционалното програмиране почти няма функционални езици без свързани променливи. Решението, което гарантира хем удобство, хем ефективност, е следното: ще предоставим на програмистите удобни езици със свързани променливи, но на ниско ниво ще реализираме тези езици посредством бързи виртуални машини без свързани променливи, а преводът ще се прави от компилатора напълно автоматично.

Първият и изключително остроумен начин за елиминация на свързаните променливи от λ-смятането всъщност е измислен преди да се

*При цилиндричните алгебри няма функционални символи, но от гледна точка на изразителната сила това не е съществено ограничение.

**В последно време скоростта на компютрите при последователни пресмятания лека полека започва да достига своя максимум. Все още обаче има възможност за увеличаване на паралелността. Тъй като релационните бази данни могат да използват в пълнота паралелността на компютрите, това означава, че през следващите години можем да очакваме, че бързината и възможностите на специализираните системи за управление на бази данни ще се увеличат в значително по-голяма степен, отколкото бързината на изпълнение на обичайните компютърни програми.

появи самото λ-смятане. Това е *комбинаторната логика* на Мойсей Шейнфинкел и Хаскел Къри (1924 – 1930 г.). На нейна база много покъсно е измислена т. н. SK-*машина*, която се използва за компилация напр. на езиците SASL и миранда. Освен оригиналния метод на Шейнфинкел и Къри, вече са разработени и много други методи за отстраняване на свързаните променливи. На практика всяка ефективна виртуална машина, използвана за реализация на съвременен функционален език (напр. категорната абстрактна машина, виртуалните машини с комбинатори и суперкомбинатори), е получена в резултат на теоретични логически изследвания.

* * *

Въпреки че като цяло уменията на компютрите да измислят математически доказателства са несравнимо по-слаби от уменията на един професионален математик, има една област, в която компютрите нерядко се справят значително по-добре от хората — доказателства за коректност на сложни паралелни програми.

Работата на една паралелна програма протича недетерминистично и за да можем да кажем, че тя е коректна, трябва да сме сигурни, че при всички възможни начини на работата програмата работи вярно. Броят на начините за протичане на изчисленията е огромен и е непосилно човек да провери ръчно всеки един от тях. Затова ако една програма е паралелна по сложен начин, програмистите няма да бъдат в състояние да се уверят, че тя е вярна. Не случайно в съвременната програмистка практика въпреки нарастващата паралелизация на съвременните компютри, съществено паралелни програми почти не се използват.

Въпреки че отказът от паралелизация е донякъде допустим подход при програмиране, той е напълно непрактичен когато се разработват интегрални схеми. Изчислителните процеси в една интегрална схема задължително трябва да бъдат силно паралелни, защото в противен случай интегралната схема би работила твърде бавно.

Решението на проблема се състои в използването на т. н. *проверка на модели* (на англ. model checking). Първо задачата, която искаме да решим, се формулира на някакъв логически език, а след това проверяваме, че интегралната схема или програмата удовлетворява логическата формулировка. Една от най-използване за целта логики е *линейната темпорална логика*. Тази логика е удобна, защото верността при нея може да се проверява посредством сравнително бързи алгоритми, използващи крайни автомати.* В днешно време няма разработчик на

*Под „сравнително бързи“ тук се има предвид с експоненциална сложност. При този вид задачи сме се примирили да считаме експоненциалните алгоритми за бър-

интегрални схеми, който не използва проверка на модели. Що се касае до използването на този метод за проверка на софтуер, то тук приложенията са засега по-ограничени и свързани най-вече с проверка на драйвери на физически устройства.

Трябва да отбележим, че проверката на модели е нещо напълно различно от т. н. доказателствено програмиране, за което ще спомена малко по-нататък. Тъй като далеч не всяка задача, която искаме да решим, може да се формулира на езика на линейната темпорална логика, методът проверка на модели е приложим само за един сравнително тесен клас задачи. Ако обаче той е приложим, то той със сигурност ще ни даде еднозначен отговор дали програмата е вярна или не. За разлика от проверката на модели, доказателственото програмиране е приложимо при всякакъв вид задачи, но не винаги дава отговор.

* * *

Различни видове *логики за знания* могат да се използват, за да се доказва коректност и безопасност на комуникационни протоколи, на възстановяване на база данни след аварии и др. Да разгледаме един хипотетичен пример. Хакерите Чингиз и Атила решили да атакуват уеб-сайта на Демагог. За да бъде успешна атаката им, те трябва да я започнат едновременно. Как могат да се договорят за това? Чингиз праща съобщение до Атила, в което съобщава часа на атаката и му пише, че няма да атакува, ако не получи потвърждение. Атила отговаря положително на запитването на Чингиз и също му пише, че ще участва в атаката, ако знае, че в нея ще участва и Чингиз. Ще осъществят ли атаката двамата? Не, защото Атила не знае дали Чингиз ще атакува и затова Атила няма да атакува. От друга страна Чингиз знае, че Атила не знае дали Чингиз ще атакува и затова знае, че Атила няма да атакува и значи и той няма да атакува. За да бъде преодолян този проблем, Чингиз пише до Атила, че знае, че Атила знае, че Чингиз иска да атакува. Но и след това потвърждение Чингиз все още не може да атакува, защото не знае дали съобщението му е било получено от Атила. Затова Атила пише до Чингиз, че знае, че Чингиз знае, че Атила знае, че Чингиз иска да атакува. Но тъй като в този момент той не знае дали Чингиз знае, че Атила знае, че Чингиз знае, че Атила знае, че Чингиз иска да атакува, то атаката отново е невъзможна.

Очевидно по този начин Чингиз и Атила не могат да започнат координирана атака. Има ли друг, по-смислен протокол, посредством който двамата могат да постигнат желаната договорка? Оказва се, че не. С използването на логики за знания може да се докаже, че колкото и

зи.

да са хитри двамата, е невъзможно да се уверят, че всеки от тях знае каквото трябва да знае.

Съждителната динамична логика с номинали е един от най-мощните логически езици, за които е известен алгоритъм за проверка на верността. С тази логика може да се решават всички задачи, които могат да се решават посредством линейната темпорална логика, но за разлика от нея, тя може да се използва и като логика за знания с цел да се проверява коректността на комуникационни протоколи. Тя е открита от Соломон Паси и проф. Тинко Тинчев, а Георги Гаргов е установил, че за тази логика съществува алгоритъм, който проверява дали една логическа формула е вярна. Това е едно от онези български математически постижения, които са цитирани най-често в международните научни публикации.

Използвайки съждителната динамична логика с номинали, Борислав Ризов е разработил система, с помощта на която филолози могат да откриват думи, притежаващи определени семантични свойства. Например те могат да зададат на системата следния въпрос: „намери ми дума, която в българския език е синоним на думата 'обитавам', но ако преведем тази дума и думата 'обитавам' на английски, ще получим думи, които не са синоними“.[*]

* * *

През последните години бурно развитие имат различни логики, описващи взаимодействията между различни геометрични обекти, в които базовото геометрично понятие не е „точка в пространството“ а „област (регион) в пространството“. Такива логики могат да се използват при разработката на „умни“ устройства, които могат да се придвижват в пространството без да бъдат управлявани от човек. Тези логики се използват също и при разработката на „изкуствения интелект“ на някои видове компютърни игри. Няколко български математици са работили върху „безточковите“ геометрии.[**] Например Владислав Ненчев е изобретил безточкови геометрични логики, в които могат да се формулират сложни съждения от типа „докато обект A се допира до B или е в контакт с C, A ще се намира изцяло в региона D“. Освен това Ненчев е намерил и ефективен алгоритъм, посредством който е възможно да се проверява верността на съждения, изказани в тези логики.

[*] Достъп до системата може да се получи на адрес `http://dcl.bas.bg/bulnet/`.

[**] Сред тях са Николай Белухов, проф. Димитър Вакарелов, проф. Георги Димов, Татяна Иванова, Владислав Ненчев, проф. Тинко Тинчев.

Теория на множествата

Компютърните програми се пишат на езици с точен синтаксис и общо взето точна семантика. Независимо по колко сложен начин са свързани по между си компонентите на една програма, този начин е описан по най-прецизен начин в текста на всяка една компютърна програма. А този програмен текст не се появява от нищото, ами преди изобщо да сме в състояние да започнем да го пишем, е нужно да отделим не малко време за мислене. И колкото по-сложна е една програма и по колкото по-сложен начин са свързани по между си компонентите ѝ, толкова по-сложна и решаващо важна е тази първа стъпка от създаването на програмата — обмислянето на нейната архитектура, на потоците от данни в нея, на взаимодействието на програмата с външния свят, на взаимодействието на един програмен компонент с друг, на структурите данни и алгоритмите, които ще се използват, на методите за откриване на грешки, на възможностите за бъдещо развитие на програмата. Ако по време на това предварително обмисляне допуснем грешка, последствията са сериозни и нерядко непоправими. Програмата трябва или да се пренапише, или да се остави с дефекти, описани в документацията ѝ.

По време на предварителното обмисляне на една програма се налага да работим с много по-абстрактни понятия, отколкото когато дойде време да я пишем. Начинът, по който разсъждаваме, е същият, който използваме тогава, когато измисляме нова математическа теория. А всеки работещ математик е познал от собствен опит, че математическата интуиция може да заблуждава. След като измисли нещо ново, математикът задължително го проверява — дава прецизни дефиниции на понятията и формулира разсъжденията, които дотогава са съществували само в неясен вид в ума му. Ако всичко излезе както трябва — хубаво, но нерядко се оказва, че след точна формулировка нещата излизат не точно такива, каквито си ги е мислил математикът.

Същото се случва и при предварителното обмисляне на една програма. И колкото по-рано открием допуснатите грешки, толкова по-добре. Само че докато математикът е трениран да използва точен математически език и точни методи за разсъждение, то проектантът на една програма за съжаление най-често е самоук и не знае как провери предварителните си идеи. На него не му остава нищо друго, освен да пристъпи към написването на програмата и попътно да си доизяснява нещата и да ги поправя, ако все още не е твърде късно.

В университетските математически курсове на студентите се преподават доказателства. На пръв поглед това може да се стори ненужно — защо например е нужно в курса по математически анализ да ни се

преподава доказателството на правилото на Гийом Франсоа дьо Лопитал? Самото правило се помни лесно и се използва лесно, а хиляди математици преди нас са чели доказателството му и са се уверили, че то е вярно. Защо трябва и ние да учим това доказателство, нима се съмняваме във верността му? Е, някога математиката се е преподавала без доказателства, а просто като съвкупност от проверени практически правила, но за щастие от тогава сме се поучили, че така не бива. В областта на програмирането обаче, още не е дошло времето да си извлечем аналогична поука.

В древността е имало велики архитекти, проектирали сгради, на които и днес се възхищаваме, но не е имало наука архитектура. За да бъдеш добър архитект се е искало огромно количество талант, изкуство, усет, опитност, вярна интуиция. Чак когато натрупаният опит бил оформен във вид на систематизирано знание, станало възможно и по-обикновени хора да станат архитекти или строителни инженери и да проектират сгради, които няма да паднат преди да са построени. Аналогична е ситуацията и със софтуерната архитектура и софтуерното инженерство. За съжаление обаче опитите за систематизиране на знанията в тези области са сравнително малко, така че добрите софтуерни архитекти и софтуерни инженери са такива не защото са завършили специалност софтуерно инженерство в някой университет.

Съвременната ситуация с обучението по информатика не е добра — в много университети студентите изобщо не се обучават как да проектират правилни компютърни програми. Езиците за спецификация, ако изобщо се преподават, се преподават сравнително повърхностно, а най-лошото е, че студентите така и не разбират кога и как трябва да се използват тези езици на практика. Немалко специалисти например препоръчват в книгите си използването на формални спецификации като вид споразумение между клиентите и програмистите, а няма съмнение, че те точно това преподават и на студентите си. Въпреки че в някои случаи това наистина е полезно, най-често формалната спецификация от една страна не отговаря на истинските желания на клиента, а от друга — забранява на програмистите да реализират варианти, които са хем по-лесни за програмиране, хем по-полезни за клиента.

В други случаи се създава погрешното впечатление, че е най-добре най-напред да се специфицира цялата програма и едва след това да се пристъпи към писане на програмния код. В действителност ако програмата е достатъчно сложна, това е наистина абсурден начин за програмиране. То е все едно учените и инженерите, участващи в проект за изпращане на хора до Луната, директно да се опитват да създадат чертежите на необходимата ракета и космическия кораб, както и плана

на полета, без преди това да са осъществили някоя по-проста част от целия проект. Полетът до Луната би бил невъзможен, ако преди това разработчиците не бяха усвоили излизането със скафандър в открития космос, маневрирането в околоземното и окололунното пространство, скачването на два апарата в околоземното и окололунното пространство и т. н. По същия начин трябва да се постъпва и при програмиране. Колкото и внимателно да сме подготвили спецификацията на програмата, неминуемо реалното писане на код ще ни поднесе изненади. Затова след като някоя част от програмата ни се изясни и сме готови със спецификацията ѝ, най-добре е да пристъпим към програмирането ѝ, макар останалата част от програмата все още да остава неясна.

За съжаление доброто желание не винаги е достатъчно... Много е лесно ако използването на език за спецификации бъде наложено силово от ръководството на една фирма, а екипът няма нужния опит, вместо този език да стане мощно средство за подпомагане на програмистите, той да се превърне в инструмент за мъчение.* Наистина има какво да се сбърка — кои свойства на програмата е важно да бъдат включени в предварителната спецификация и за кои ще бъде вредно това да се прави, до каква степен да проявяваме гъвкавост и да модифицираме предварителната спецификация, ако след като започне писането на програмата установим, че нещо е било по-добре да се направи по друг начин и т. н. И всички тези проблеми са при положение, че екипът вече е усвоил използването на формален език за спецификации, което само по себе си е предизвикателство.

* * *

Вече споменахме, че използването на формална спецификация може да помогне на проектантите да открият фундаментални грешки и недостатъци в конструкцията на програмата много преди да започне нейното писане. Тази полза е налице дори тогава, когато проектантът е един човек и прави спецификацията само за себе си. И наистина, формалното описание на нужните свойства отнема много по-малко време, отколкото самото написване на програмата, а откритите благодарение на него грешки се поправят много по-бързо. И когато наистина дойде време да пишем програмния текст, структурата на програмата ни е ясна и затова програмираме по-бързо и допускаме по-малко грешки.

От използването на формални спецификации произтича и една друга, по-незабележима, но съвсем не маловажна полза. Каквато и дейност да извършва човек, той неволно винаги се опитва да я свърши

*Един познат програмист ми каза точно това — езиците за спецификация са инструмент за измъчване на програмистите и заблуждаване на клиентите.

по възможно най-лесния начин. Затова когато програмистите пишат програмния текст, те разсъждават локално — как да свършат задачата по най-простия и елегантен начин без да мислят дали създадените по този начин библиотеки от класове или функции са прости за използване и с ясно и разбираемо действие. Когато обаче се пише спецификация, човек пак в желанието да си спести труда, не мисли колко трудна е реализацията, а само за това колко просто са формулирани желаните свойства на програмните компоненти. В резултат на това взаимодействието на програмните компоненти става възможно най-опростено и лесно за разбиране, а програмните грешки най-често са локални и лесни за оправяне. Опростеното взаимодействие на програмните компоненти помага и при последващата поддръжка на програмния код. Когато модифицират вече написаната програма, програмистите, дори да не са участвали в първоначалното ѝ написване, се чувстват много по-уверени, че действията им няма да доведат до непредвидени грешки.

Фирми, които са опитвали да специфицират вече написани програми, са установили, че това е безсмислена задача, защото изведнъж става нужно да опишат точно всичката онази невидима сложност във вече написаната програмата. [2, стр. 16] Сложност, която със сигурност е затруднила поправянето на програмните грешки и със сигурност ще затрудни последващата поддръжка.

* * *

Къде във всичко това присъства теория на множествата? Е, оказва се, че всички езици за формална спецификация по един или друг начин се свеждат до използване на езика на теория на множествата. Тази незаменимост на езика на теория на множествата не е изненадваща — щом като този език е от една страна необходим в съвременната математика, а от друга страна напълно достатъчен, за да изкажем на него цялата съвременна математика, без нито едно изключение, защо да не очакваме същото да важи и за програмирането?*

По-старите езици за спецификация (напр. зед) умишлено подчертават връзката си с теория на множествата. Използването на тези езици естествено е невъзможно от програмисти, които не са получили нужната математическа подготовка. В по-ново време обаче стана ясно, че вместо непосредствено за множества, можем да използваме терминология, по-разбираема за програмистите. Например вместо да казваме, че x е елемент на множеството A, можем да казваме, че x е обект от

*Дълго време изглеждаше, че конструктивната математика задължително изисква използването на не-множествен език. След създаването на конструктивната теория на множествата стана ясно, че това не е така.

класа A; вместо да казваме, че предикатът $p(x, y)$ е истина за x и y, можем да казваме, че обектът x има поле с име p, чиято стойност е y; вместо да използваме квантори, може да *скулемизираме* и т. н. Всичко това обаче в никакъв случай не е заместител на теория на множествата, а просто начин да се даде на програмистите без математическа подготовка език, равносилен на езика на теория на множествата. При използването на такъв език е важно да се знае, че „класовете“, за които се говори в спецификацията, изобщо не е нужно да съответстват на реални класове в програмата, „полетата“, които притежават обектите, не е нужно да съществуват и в истинската програма и т. н.

Теория на изчислимостта

Когато на света се появили компютрите, оказало се, че важна част от теорията на това, какво и как може да се смята с тях, вече съществувала. Днес този дял от математическата логика се нарича „*теория на изчислимостта*“.*

В доказателството на своите теореми за непълнота Гьодел дефинирал понятието „*примитивнорекурсивна функция*“. Малко по-късно, през 1934 г., Гьодел дефинирал и понятието „*общорекурсивна функция*“, което се оказало еквивалентно на съвременното понятие „изчислима функция“.** По онова време обаче Гьодел все още не допускал, че понятието, което той дефинирал, обхващало всички функции на естествени числа, за които може да се счита, че са изчислими.

Тезата, че интуитивното понятие „изчислима функция“ отговаря на даден, точно дефиниран клас от математически функции, е изказана за пръв път през 1936 г. от трима математици — Аланзо Чърч, Алън Тюринг и Емил Пост. Всеки от тях дал различна дефиниция на това какво значи изчислима функция, но тези дефиниции се оказали еквивалентни както по между си, така и на по-ранната дефиниция на Гьодел. Според Чърч изчислимите функции са точно функциите, които могат да се дефинират с т. н. λ-термове. Това твърдение днес е известно като „*тезис на Чърч*“, а създаденото от Чърч *λ-смятане* служи за математическа основа на почти всички съвременни функционални езици. Тюринг пък дефинирал това, което днес е известно като „машина на Тюринг“, а твърдението, че изчилимите функции са точно функциите, които може да се пресмятат с машина на Тюринг, е известно като

*Дълго време този дял се наричал „теория на рекурсията“, а изчислимите функции — рекурсивни функции.

**Според Гьодел идеята за тези функции му е била подсказана от Ербран.

„*тезис на Тюринг*“.* Дефиницията на Пост за изчислима функция е удивително сходна с дефиницията на Тюринг, макар да е получена напълно независимо. Простотата на тези две дефиниции ги прави много удобни за математически изследвания.

По-късно се появили и други, различни от тях дефиниции на „изчислима функция“, но и те се оказали еквивалентни по между си. Сред тях по-важни са каноничните системи на Пост (1943), нормалните алгоритми на Марков (1948) и различните видове регистрови машини (1954 – 1967). Каноничните системи на Пост и нормалните алгоритми на Марков стоят в основата на символните пресмятания в съвременната компютърна алгебра, както и на някои езици за програмиране, напр. рефал. Регистровите машини пък и особено машините на Мински съчетават теоретичните удобства, които имат машините на Тюринг, с обичайния стил на програмиране, който се използва при императивните езици за програмиране.

В някои случаи теорията на различните изчислителни модели дава идеи за нови стилове за програмиране (напр. идеята на функционалното програмиране е дошла от λ-смятането). В други случаи тя дава идеи за ефективно използване на виртуални машини. Методи от математическата логика се използват също и при разработката на компилатори и особено на оптимизиращи компилатори. Все по-голямо значение добиват системите за статичен анализ на програмен код. Такива системи например автоматично могат да откриват дали някоя променлива се използва без да е инициализирана.

* * *

В теория на изчислимостта се разработват различни методи за дефиниране на семантиката на различните езици за програмиране. Наличието на точна семантика на един език е много важно при реализацията му, защото в противен случай комбинираното използване на различни свойства на езика, всяко от които само по себе си изглежда ясно, се оказва, че води до програми, за които не е ясно какво точно трябва да правят, и затова компилаторът работи неправилно.

Различните изчислителни модели и семантиките на езиците за програмиране могат да подскажат на създателите на нови езици за програмиране как да дефинират езиците така, че те да бъдат ясни и с проста семантика (в противен случай става много трудно да се пишат верни програми на тези езици). Например в областта на обектноориентираното програмиране използването на сходна терминология от различните

*Тъй като дефинициите на Чърч и Тюринг са еквивалентни, от математическа гледна точка тезисът на Чърч и тезисът на Тюринг казват едно и също нещо.

езици за програмиране създава илюзията, че всички те реализират на практика едно и също нещо. В действителност на семантиката на различните обектноориентирани езици показва, че те могат да се разделят на две групи. В едната група спадат езици, чиято математическа теория се базира на безтипови обекти. В тази група попадат например езиците смолтоук и обджектив си.* При множественото наследяване при тези езици най-много един от родителските класове може да бъде истински, докато останалите могат да предлагат само интерфейс, но не и конкретна реализация. Втората група обектноориентирани езици са тези, чиято математическа теория се базира на някоя теория на типовете. Към тази група езици спада айфел. При множественото наследяване при тези езици често се допуска всеки един от родителските класове да предлага конкретна реализация. А има и езици като си++, при чието създаване явно не се е обръщало достатъчно внимание на това каква точно трябва да бъде тяхната семантика. Може би затова множественото наследяване при си++ е трудно за използване и всъщност никой не го използва.**

В днешно време има много езици, които изобщо не притежават оператор goto, но това, че операторът goto наистина не е нужен, става ясно от една теорема на Корадо Бьом и Джузепе Якопини от 1966 г.

Важни резултати в областта на семантиките на езиците за програмиране са получени в България от проф. Иван Сосков. По-горе се спомена, че съществуват различни дефиниции на понятието изчислима функция и всички те се оказват еквивалентни. Обаче това е така само когато говорим за функции върху естествени числа. Затова един естествен въпрос е какво се случва ако вместо функции върху естествени числа използваме функции, дефинирани за произволен абстрактен тип данни. Оказва се, че в този случай различните математически модели за изчислимост не са еквивалентни. Така например Сосков е установил, че: 1. за всяка императивна програма можем да намерим еквивалентна на нея функционална програма, но с функционални програми понякога може да се правят и неща, които не могат да се направят с императивни [28] и 2. за всяка функционална програма можем да намерим еквивалентна на нея логическа програма, но с логически програми

*Към тази група езици спада и джава, въпреки че езикът създава илюзията за статична типизация.

**Понякога хора, повлияни от си++, твърдят, че множественото наследяване е едва ли не безполезно и опасно, когато повече от един от родителските класове предлага реализация. Всъщност езикът айфел нагледно показва точно обратното — ако езикът е базиран на хубава математическа теория, такова наследяване не само е напълно безопасно и полезно, но също така и много приятно за използване.

понякога може да се правят и неща, които не могат да се направят с функционални. [17]

* * *

Един от важните проблеми в съвременната информатика е свързан с удобното и безопасно използване на паралелни пресмятания. Този проблем има голямо практическо значение, защото компютрите стават все по-паралелни. За решаването му се разчита на създаването на прост математически модел на паралелните пресмятания, но за съжаление такъв все още не е измислен. Вредата от това е не само теоретична — отсъствието на хубава математическа теория води до това, че паралелното програмиране с право се счита за опасно и на практика почти винаги води до трудни за откриване и оправяне програмни грешки. Два съществуващи модела на паралелните пресмятания се дават напр. от *π-смятането* и *джойн-смятането*. Макар те да са по-сложни, отколкото ни се иска, езиците за програмиране, в които средствата за паралелизъм са базирани на някое от тези две смятания, са по-удобни за писане на правилни паралелни програми, отколкото езиците, при които паралелизмът не е базиран на никаква теория.[*]

Скоро след появата на компютрите станало ясно, че има разлика между теоретично изчислима функция и функция, която е изчислима на практика. Това мотивирало бързото развитие на изключително важната теория на сложността. За съжаление в тази област има много задачи, които се оказват изключително трудни и все още остават нерешени. Най-знаменитата и все още не решена задача е въпросът дали „P=NP“.[**] Тук P обозначава функциите, които грубо казано може да се пресмятат за полиномиално време на последователен компютър, а NP — функциите, които може да се пресмятат за полиномиално време на неограничено паралелен компютър. Този проблем е важен по две причини. От една страна класът P съдържа функциите, за които можем да считаме, че се смятат за разумно кратко време на компютър,[***] а от

[*] Изводът, който можем да си направим, е следният: програмирането с лоша математика е по-добро от програмирането с никаква математика.

[**] Обявена е награда от 1 000 000 долара за този, който пръв намери вярно решение на този проблем.

[***] Става въпрос за компютър със стандартна архитектура. Квантовите компютри могат да решават ефективно някои задачи, които изискват неограничена паралелизация. В частност повечето от използваните в момента алгоритми за асиметрична криптография ще престанат да бъдат сигурни, ако разполагаме с квантови компютри с няколко хиляди кубита памет. Предполага се обаче, че не всяка NP задача може да се решава ефективно от квантов компютър, и има алгоритми за асиметрична криптография, които са използваеми с нормален компютър и за които се счита, че биха останали сигурни дори и да разполагаме с квантови компютри с голяма

друга — има много на брой изключително важни за практиката задачи, които принадлежат към класа NP. Ако имаме щастието да се окаже, че P=NP, то това означава, че ще разполагаме с ефективен алгоритъм за решаване на тези важни задачи. От друга страна, ако имаме нещастието да се окаже, че P=NP, то това може да направи неизползваеми всички използвани на практика криптографски алгоритми.

Освен с директно измерване на времето за пресмятане и използваната памет, един алтернативен подход за измерване на сложността на една изчислима функция е това колко сложна е нейната дефиниция (на пръв поглед не се вижда защо има връзка между сложността на дефиницията на една функция и времето за нейното пресмятане, но такава има). Функциите, които притежават проста дефиниция, се изучават в теорията на субрекурсивните функции. Субрекурсивните функции описват по-добре отколкото изчислимите функции това, какво на практика може да се пресмята с компютър. Затова е важно да бъде изследвана не само субрекурсивната изчислимост между естествени числа, но също и субрекурсивната изчислимост между други полезни математически обекти. Субрекурсивната изчислимост на реални числа все още е сравнително малко изследвана област, като повечето от съществуващите резултати са получени в България от Иван Георгиев и проф. Димитър Скордев.

Когато в математиката бъде разработена теорията на определен вид, в някакъв смисъл подобни обекти, е естествено тази подобност да бъде формализирана, след което систематично да бъдат потърсени други обекти, притежаващи така формулираните свойства. Например в алгебрата след като се открият общите свойства на различните числови полета, получаваме аксиомите на абстрактното понятие поле и установяваме, че има полезни нечислови полета. В областта на теория на изчислимостта същото нещо е направено за пръв път от проф. Димитър Скордев [27, 16]. Той е дефинирал аксиоматично понятието „*комбинаторно пространство*“ и е показал че някои от основните резултати в теория на изчислимостта могат да бъдат изведени като следствие от аксиомите на комбинаторните пространства. Самите комбинаторни пространства пък се оказва, че имат много интересни модели, които ни дават неочаквани нови видове изчислимости (много от тях все още очакват някой да им разработи теорията в пълнота). След комбинаторните пространства Любомир Иванов [9] е дефинирал понятието „*оперативно пространство*“, а важни резултати в тази област са получени и от Йордан Зашев. Всички тези резултати са дали началото на това,

памет.

което днес се нарича аксиоматична теория на изчислимостта.*

Теория на доказателствата и конструктивна математика

Вече бе споменато, че ако искаме да разработваме математиката конструктивно, ще се наложи да разсъждаваме, използвайки алтернативна логика, която е различна от класическата. Тази логика се нарича *интуиционистка логика*, а законите ѝ били формулирани от Аренд Хейтинг през 1930 г.**

В класическата математика когато твърдим, че нещо съществува, е все едно дали разполагаме с метод, посредством който можем да намерим съществуващото нещо. В конструктивната математика това не е така. Ако един конструктивист докаже, че съществува обект с определено свойство, то значи той разполага с метод, посредством който може да бъде намерен този обект. Ако пък бъде доказано съждение от вида „за всяко x е вярно $x \leqq 0$ или $x \geqq 0$“, от тук автоматично ще следва, че има метод, с помощта на който за всяко x можем да познаваме дали е вярно $x \leqq 0$ или $x \geqq 0$.

Според конструктивистите доказателството на едно съждение A представлява точно определена конструкция, която реализира A. Например доказателството на твърдение от вида „ако A, то B“ ни дава конкретен метод, с помощта на който ако разполагаме с конкретно доказателство на A, ще получим конкретно доказателство на B. С други думи можем да си мислим, че доказателството на „ако A, то B“ е функция, на която ако ѝ дадем като аргумент доказателство на A, ще ни върне като стойност доказателство на B. Доказателството пък на твърдение от вида „A и B“ представлява просто комбинацията от конкретно доказателство на A и конкретно доказателство на B.***

Казаното до тук може би подсказва, че би трябвало да има някаква връзка между интуиционистката логика и програмирането. Дали няма да се окаже, че всяка интуиционистка логическа формула дефинира ня-

*Това засега може би е единственият дял в математиката, който е създаден в България. В България той по-често се нарича „алгебрична теория на рекурсията“.

**Това, че е възможно формулираме конструктивните разсъждения, използвайки сравнително прости формални правила и аксиоми, е било изненада за създателя на интуиционизма Брауер. Отначало той считал, че не някакви формални правила и аксиоми, а единствено ясната математическа интуиция може да ни даде правилна основа на математиката. Затова той се е противопоставял на опитите за формализация на интуиционизма, а изследванията на Хейтинг наричал „безрезултатни упражнения“.

***Този начин за интерпретация на логическите връзки е известен като интерпретация на Брауер-Хейтинг-Колмогоров.

каква изчислителна задача, а доказателството на тази интуиционистка формула представлява програма, която решава тази изчислителна задача? Оказва се, че това е точно така и това е забелязано от Хаскел Къри и Уилям Алвин Хауърд. Може да считаме, че всяка интуиционистка формула представлява някакъв тип данни, а всяко доказателство на формулата е програмен обект (напр. функция), притежаващ съответния тип. Вярно е и обратното — може да си мислим, че типовете данни са интуиционистки формули от определен вид, а всяка компютърна програма представлява някакво интуиционистко доказателство. Наборът обекти, дефинирани в една компютърна библиотека, може да бъде считан за набор от аксиоми на интуиционистка математическа теория, а всяка програма, която е написана, използвайки единствено обектите от предоставената библиотека, представлява интуиционистка теорема, доказана използвайки само дадените аксиоми.

Това удивително съответствие между програмиране и математика е известно като „*изоморфизъм на Къри – Хауърд*“. То може да бъде изказано не само неформално (както направихме тук), но също и като точно математическо твърдение. Изоморфизмът на Къри – Хауърд показва, че програмирането и правенето на математика представляват не просто две подобни дейности, а напълно идентични дейности.

Ползата от този изоморфизъм е не само философска, защото той прояснява какво всъщност представляват типовете данни в езиците за програмиране. Създателите на почти всички типизирани функционални езици за програмиране умишлено разчитат на този изоморфизъм, за да си гарантират, че системата от типове ще бъде хем проста за използване, хем пълна. Напълно заслужено на името на Хаскел Къри е наречен един от най-популярните в момента езици за функционално програмиране — хаскел.

Този изоморфизъм има и други практически приложения. Например той подсказва как да разширим типовете данни в един език за програмиране по такъв начин, че спецификацията на това какво прави една функция да бъде част от нейния тип. И също както при използването на обикновени типове компилаторите могат статично да проверяват дали няма нарушения на типовете, така и при използването на тези разширени типове се оказва, че това е възможно. Това позволява компилаторът не само да компилира програмата, но и да удостоверява, че тя е вярна и не съдържа нито една програмна грешка. Въпреки че засега всички съществуващи системи от този тип са трудни за използване, вече има оптимизиращ компилатор на си, за който знаем, че няма програмни грешки, защото е програмиран, използвайки система с интуиционистки типове.

В България изследвания свързани с изоморфизма на Къри – Хауърд са правени от Трифон Трифонов.

* * *

От напълно различен вид е връзката между математика и програмиране при логическото програмиране. При логическото програмиране логическите формули се интерпретират не като типове данни (както е при функционалното програмиране), а като компютърни програми. Процесът на търсене на доказателство на логическата формула пък представлява изпълнението тази програма.

Освен като теоретична основа на логическото програмиране, компютърното търсене на доказателства е полезно и само по себе си. Наистина, съществуващите в момента алгоритми за автоматично доказателство на теореми не могат да се сравняват по ефективност с уменията на един професионален математик. Всеки, който се е занимавал професионално с програмиране, обаче знае, че през повечето време на програмистите им се налага да пишат напълно тривиален и безинтересен от математическа гледна точка програмен код. Не може ли „отговорността“ за този тривиален програмен код да бъде поверена на компютрите, а за хората да останат само по-интересните и интелектуално предизвикателни задачи?

На този въпрос може да бъде отговорено двояко. От една страна — да. Съществуват системи, които се справят много добре с генерирането на програмен код. Но на практика — не. Причината е тази, че да обясним на компютъра какво точно искаме да направи дадена програма може да бъде не по-малко досадно, отколкото просто да седнем и да си напишем сами програмата. Въпреки това изследванията за намиране на удобна за използване система за автоматично генериране на код продължават.*

Съвсем по-друг начин стоят нещата при доказателственото програмиране, т.е. тогава, когато се пишат програми, за които искаме да сме сигурни, че не съдържат грешки. До голяма степен именно благодарение на успехите в областта на автоматичното доказателство на теореми доказателственото програмиране буквално през последните няколко години от теоретично академично упражнение се превърна в нещо, което е напълно използваемо на практика. Фирмите, които се занимават с писането на програми без грешки, са установили, че дори и при много големи програми обикновено е достатъчно в екипа да има само един математик логик, който да установява коректността на онези места в

*Когато такава система бъде измислена, навсякъде ще се търсят логици, а програмистите ще останат без хляб.

програмата, чиято коректност не е могла да се докаже автоматично от компютър [2, стр. 376]. Интересно е, че системите за доказателствено програмиране могат да формулират задачите, които не са успели сами да докажат, по такъв начин, че не е нужно математикът логик да умее да програмира и всъщност не е нужно дори да знае за какво служи програмата, от която е възникнала поставената задача.

1.3. Логическо програмиране

Създаването на пролог

През 1958 Франция била разкъсвана от политически и икономически проблеми. Остров Корсика бил завлядян от алжирски военни и имало реална опасност от държавен преврат. В обществото нямало единство, политическата власт била слаба. Тогава в политиката се намесил генерал Шарл дьо Гол, герой от Втората световна война. Генерал Дьо Гол решил, че за да се решат проблемите, френското общество трябвало да бъде по-обединено, а единственият начин това да стане, е да го поведе под патриотични лозунги. Затова той заявил: „Нашата страна пред лицето на другите страни трябва да се стреми към велики цели и да не се прекланя пред никого, защото в противен случай може да се окаже в смъртна опасност“. Франция променила коренно политиката си. Въпреки ненавистта си към комунизма, Дьо Гол изкарал Франия от НАТО и дори започнал политическа заигравка със СССР. Използвайки противопоставянето между тогавашните суперсили САЩ и СССР, Дьо Гол успял да увеличи значително международната значимост на Франция и правото ѝ да води независима политика.

През 1967 пред стохиляден митинг в Монреал Шарл дьо Гол се провикнал: „Да живее свободен Квебек!“. Отношенията между Франция и Квебек достигнали ниво подобно на това, което имат независимите държави. Франция отзовала своя „Генерален консул в град Квебек“ и вместо него назначила „Генерален консул пред правителството на Квебек“. Започнали взаимни визити на френски и квебекски политици. Френските политици редовно нарушавали дипломатическите правила като посещавали Квебек без изобщо да се появят преди това в канадската столица Отава. Разширило се научното сътрудничество между Франция и Квебек. Френски учени се премествали да работят и преподават в Квебек, а квебекски — във Франция.

През 1967 г. в Монреал, Квебек, се преместил и младият френски учен Ален Колмеро. Тук той създал Кю-системите — един от първите формализми за обработка на естествени езици. Колмеро бил научен

ръководител на квебекчанина Жан Трудел, който работил в областта на автоматичното доказателство на теореми. През 1970 г. Колмеро поканил в Монреал и току-що дипломиралите се французи Робер Пазеро и Филип Русел. Там те се запознали с компютърната обработка на естествени езици.

През 1970 г. Дьо Гол починал и година по-късно групата около Колмеро заедно с квебекчанина Пазеро се върнала във Франция. Тук те създали система, която можела да води диалог на естествен език, като сама правела логически изводи. Езиковата част била разработена от Колмеро и Пазеро, а за автоматичното правене на логически изводи отговаряли Трудел и Русел. Още през същата година системата била достатъчно развита, за да води следния диалог:

Потребител:

Котките убиват мишки.

Том е котка, която не обича мишки, които ядат сирене.

Джери е мишка, която яде сирене.

Макс не е мишка.

Какво прави Том?

Компютър:

Том не обича мишки, които ядат сирене.

Том убива мишки.

Потребител:

Кой е котка?

Компютър:

Том.

Потребител:

Какво яде Джери?

Компютър:

Сирене.

Потребител:

Кой не обича мишките, които ядат сирене?

Компютър:

Том.

Потребител:

Какво яде Том?

Компютър:

> Това, което ядат котките, които не харесват мишки, които ядат сирене.

През това време Русел и Трудел продължили да експериментират с различни методи за автоматично правене на логически изводи. През 1971 Трудел се запознал с току-що измислената от Робърт Ковалски и Доналд Кунър SL-резолюция. Ковалски и Кунър работели в Единбург, но Трудел убедил останалите от групата да поканят Ковалски за кратко посещение в Марсилия, за да им разкаже за метода.

Марсилската група около Колмеро се състояла от хора практици. Те не пропускали случая да се запознаят с нови теоретични резултати, които биха могли да им свършат работа, но собствената им научна дейност не била свързана със създаване на нова математика. От друга страна, Ковалски бил теоретик. Когато прилагаме една теория на практика, винаги се налага да решаваме проблеми, които са от практическо естество и не представляват теоретичен интерес. На математиците теоретици, какъвто бил и Ковалски, не им е безинтересно как се прилагат на практика откритите от тях неща, но много често на тях не им харесва сами да решават практическите проблеми свързани с приложението на теорията. Когато обаче се срещнат добри практици с добри теоретици, винаги всеки научава нещо ново и полезно за себе си. Марсилската група се запознала по-добре със свойствата на SL-резолюцията, а това се оказало решаващо важно за измислянето на пролог. Ковалски пък научил неща за практическото използване на резолюцията, които не знаел по-рано, създал теоретичните основи на логическото програмиране и в крайна сметка точно с това се и прочул. За SL-резолюцията, от която тръгнало всичко, днес малцина си спомнят, а още по-малко са тези, които знаят какво всъщност представлява тя.

През 1972 г. Ковалски бил поканен да посети Марсилия за втори път, този път за по-дълго. По това време групата вече имала по-ясна интуиция как да си представя автоматичното доказване с SL-резолюция като изчислителна процедура. Те знаели как да аксиоматизират прости неща като събиране на числа, конкатенация на списъци, обръщане на списък и т. н. по такъв начин, че SL-резолюцията да намира нужния резултат ефективно.

След заминаването на Ковалски, Колмеро измислил как да елиминира Кю-системите. Всъщност, използвайки открития от Колмеро начин, и днес на пролог можем да много лесно синтактичен разбор за произволен безконтекстен език. По този начин за пръв път започнала да изглежда възможна идеята, цялата система, която разработвали в

Марсилия, да се реализира, използвайки логика.

Взето било едно много важно решение: оказало се, че за да се „изпрограмират“ нужните неща, било достатъчно да се използват само хорнови клаузи. С цел по-добра ефективност и по-ясна процедурна семантика, те взели още едно важно решение: позволили само резолвенти с първия литерал на хорновите клаузи. По този начин от чисто практически съображения те открили това, което днес се нарича SLD-резолюция. Те предполагали, че SLD-резолюцията е непълна, но считали, че това е приемлива цена заради по-добрата ефективност. Така в края на 1972 г. се появила първата реализация на езика пролог. На следващата година (1973) Ковалски доказал, че SLD-резолюцията всъщност не е непълна [10], а през 1974 заедно с Мартен ван Емден дефинирал и семантика на логическите програми с неподвижна точка [20].*

Името „пролог“ било измислено от съпругата на Филип Русел и е съкращение от „PROgrammation en LOGique“ (програмиране посредством логика).

Приложенията на пролог

Езикът пролог впечатлява с това по какъв елегантен начин съчетава в себе си мощ и простота. Кой друг език не съдържа нито една запазена дума и при това остава удобен за използване, дори ако го лишим от всички вградени в него предикати, функции и т. н.? На мнозина от първите ентусиасти на пролог им се е струвало, че е само въпрос на време, кога пролог ще стана най-популярният език за програмиране. Някои считали, че други езици за програмиране изобщо не било нужно да се учат.

Разбира се, вече знаем, че тези представи се оказали много далеч от истината. Езикът пролог изобщо не успял да стане толкова популярен, колкото очаквали. Въпреки това има някои видове приложения, при които пролог се е доказал като един от най-удобните съществуващи езици за програмиране.

* * *

Първата група приложения на пролог са свързани с основния тип данни на езика — термовете. Ако ни се налага да извършваме сложни манипулации на дървесни структури данни, то си струва да обмислим, дали да не програмираме на пролог. Компилаторите, и особено оптимизиращите компилатори, са програми, на които им се налага манипулират такива дървесни структури. Първата реализация на езика

*Ван Емден е автор на термина „SLD-резолюция“.

ерланг била написана на пролог. Също и при езика СПАРК важна част от реализацията, която се грижи за коректността на компилираната програма, е написана на пролог.

* * *

Втората група от приложения на пролог е свързана с това, че всяка програма на пролог представлява съвкупност от правила, а в някои случаи е най-удобно да опишем действията на приложението, което искаме да реализираме, именно посредством съвкупност от правила. Ако решим да програмираме една такава програма на традиционен език, бихме получили огромно количество вложени един в друг условни оператори. Логиката на такава програма е трудна за проследаване, а грешките — невъзможни за оправяне.

На пролог е удобно да се реализират т. н. бизнес правила. Така например корпорацията Ериксон с такава система е вземала решения, свързани с продажбите на продукти, а банки и застрахователни компании са оценявали заявленията за предоставяне на кредити и очакваните печалби.

На пролог е написана една от най-популярните система за автоматизация на документи (DealBuilder). Тя може автоматично да генерира текстове на договори и други юридически документи, използвайки сложен набор от правила.

На пролог е написана система, която помага на потребителите да настроят мобилните си телефони. Тя се използва на уеб сайтовете на много мобилни оператори. Ясно е, че не е лесно да се съчетаят в прост алгоритъм специфичните изисквания на всеки един телефон със специфичните изисквания на отделните услуги, предоставяни от оператора.

Около една трета от резервациите на самолетни билети по света се извършва посредством система, написана на пролог.

В Windows NT 3.1 системата, която решава кои драйвери е най-подходящо да се заредят от ядрото, е написана на пролог.

* * *

Третата група от приложения са тези, при които програмата трябва да реагира по сложен начин на огромно количество взаимнозависими фактори.

По данни на Централното разузнавателно управление на САЩ, на пролог е написан софтуерът на руската космическа совалка „Буран“, която за пръв път извършила кацане на летище в напълно автоматичен режим (безпилотно и без радиоуправление).

На космонавтите на Международната космическа станция често им се налага да извършват сложни дейности. При тяхното извършване те трябва да си подсказват с инструкции от таблет, което отвлича внима-

нието им и губи време. Американската космическа агенция НАСА обаче е реализирала на пролог система за гласово управление на браузър. Космонавтът казва на глас каква информация иска от компютъра, а компютърът също му отговаря със синтезиран глас.

Понякога на пролог е удобно да се пишат системи за извличане на ценна информация от огромно количество неструктурирани данни (data mining). Например Американската агенция за електронно разузнаване използва пролог, за да анализира социалните мрежи и да открива хора, за които има вероятност да се превърнат в терористи.

Много от съществуващите експертни системи са написани на пролог. Някои примери за експертни системи, написани на пролог са:

- експертна система, която анализира условията, при които се отглеждат свинете в свинефермите, и предлага промени за начина на хранене на свинете, температурата, влажността и вентилацията на въздуха, използваните породи и т. н. В някои случаи тази експертна система е увеличавала печалбата до пет пъти;
- на пролог са написани няколко експертни системи, които анализират околната среда. Те се използват за прогнозиране на времето, за прогнозиране на предстоящи глобални изменения в климата, за прогнозиране на замърсеността на въздуха в градовете и др.;
- експертна система, която предлага методи за снабдяване на населението с питейна вода в случай на стихийни бедствия, производствени аварии или война;
- самолетостроителната фирма Боинг използва пролог за програма, която снабдява работниците с необходими инструкции. Тези инструкции се получават от различни източници — обикновени файлове, бази данни, обектноориентирани бази данни, други експертни системи. Достъпът до тези източници обикновено е отдалечен (по мрежата), но всичко става невидимо за потребителя. Един от разработчиците казва: „Работата по написването на машина за заявки на пролог бе минимална в сравнение с ЛИСП. А да я напишем на някой обикновен език би било немислимо, защото това на практика би означавало да измислим отново пролог“;
- „изкуственият интелект“ на някои компютърни игри използва пролог.

Логическо програмиране с ограничения

. . .
. . .
. . . *constraint (logic) programming*
. . .
. . .

Други разновидности логическо програмиране

Най-строгата дефиниция на това какво означава „логическо програмиране“ е следната:

Дефиниция. *Логическото програмиране* е метод за описание на компютърни алгоритми, при който:

- компютърната програма представлява логическа задача, формулирана на някакъв формален логически език;
- изпълнението на програмата от компютъра представлява търсене на доказателство на поставената логическа задача.

Например при езика пролог програмата представлява задача от следния вид: ако Γ е множество от хорнови клаузи, а φ е конюнкция от атомарни формули, да се докаже, че от Γ следва изпълнимостта на φ. Методът, който компютърът използва, за да реши тази задача, е SLD-резолюцията.

Ако вместо обикновени хорнови клаузи, използваме друг логически език, получаваме други езици за логическо програмиране.

* * *

Езикът мъркюри е на пръв поглед тривиално разширение на пролог — той се получава просто като добавим към езика типове данни и т. н. режими на предикатите. От логическа гледна точка добавянето на типове не дава на езика по-голяма изразителна сила. Когато обаче става въпрос за програмиране, тази проста добавка има следните две важни следствия. Първо, езикът мъркюри позволява да се пишат изключително бързи програми. Второ, за разлика от пролог, при мъркюри конюнкцията е напълно комутативна, а ако програмата не се зацикля, тогава и дизюнкцията става комутативна и съответствието между програмата и желаната логическа семантика е много по-пълно. Въпреки че при мъркюри няма нелогически предикати като предиката за отсичане на пролог, на мъркюри може да се пишат далеч по-бързи програми, отколкото на пролог.

* * *

Езикът ламбда-пролог добавя към пролог следните неща:

– Типове данни.

– Явни квантори. Доказателството на $\exists \mathtt{x}\, \mathtt{p}(\mathtt{x})$ се прави по същия начин, както и на пролог (където кванторът $\exists$ се подразбира). Доказателството на $\forall \mathtt{x}\, \mathtt{p}(\mathtt{x})$ се прави посредством *скулемизация* — компютърът добавя нов символ за константа `c`, доказва $\mathtt{p}(\mathtt{c})$ и след това „забравя“ за символа `c`.

– Импликация в целите. Доказателството на $\varphi \Rightarrow \psi$ се прави по следния начин — компютърът добавя към базата знания φ, доказва ψ и след това премахва φ. При програмиране тази възможност е удобно да се използва, за да се намали броят на аргументите на един предикат, както и за модулно програмиране.

– Термове със свързани променливи. Тъй като не е известен логически език или език за програмиране, при който формулите или програмите да не могат да се представят по естествен начин посредством термове със свързани променливи, то ламбда-пролог е вероятно най-удобният програмен език за обработка на формални езици.

Добавките, които ламбда-пролог прави към пролог не са „случайни приумици“, а се базират на интересна логическа теория.

* * *

Има няколко известни езика, които макар и да не отговарят на строгата дефиниция за „логическо програмиране“, дадена по-горе, все пак са силно повлияни от пролог и логическото програмиране.

* * *

Езикът ерланг се използва за писане на разпределени, паралелни програми, които са устойчиви на аварии и работят без да се спират. Много от най-натоварените уеб-сайтове по света използват ерланг за част от услугите, които предоставят. Първата версия на ерланг е реализирана на пролог, а синтаксисът на ерланг е силно повлиян от синтаксиса на пролог.

* * *

Тъй като езикът ес кю ел не използва свързани променливи, а алгебрични операции, той е лесен за реализация, но неудобен за използване. Най-перспективният език със свързани променливи за заявки към бази данни е дейталог. Една заявка на дейталог на пръв поглед прилича на програма на пролог. Нека например в базата данни имаме таблица $\mathtt{оценка}(x, y)$, която казва, че студентът x има оценка y по логическо

програмиране. И нека още имаме таблица `група`(x, y), която казва, че студентът x е от група y. Питаме се в кои административни групи има студенти, получили слаба оценка по логическо програмиране. На дейталог това може да бъде направено така:

```
проблем(Група) :- оценка(Студент,2), група(Студент,Група).
?- проблем(Група).
```

Същата заявка на ес кю ел включва следните операции:

- нека t_1 е таблицата, която се получава от `проблем` като отделим само редовете, в които оценката е слаба;
- нека t_2 е таблицата, която се получава от t_1 като изтрием стълба с оценките;
- нека t_3 е таблицата, която се получава като слеем таблица `група` с таблица t_2, приравнявайки стълбовете със студентите;
- нека t_4 е таблицата, която се получава от t_3 като изтрием стълба със студентите;
- изкарай като отговор таблица t_4.

* * *

Счита се, че не съществува алгоритъм, който винаги успява да решава ефективно задачи, за които е доказано, че са NP-пълни. Едно сравнително скорошно неочаквано откритие е това, че има алгоритми, които макар и не винаги, но в много случаи успяват да решават NP-пълни задачи. Тъй като има много важни за практиката задачи, които са NP-пълни, това откритие има не само теоретична, но и голяма приложна стойност. Реализирани са системи за answer set programming, които в повечето случаи използват езици, които много приличат на пролог.

*
* * *
*

За подобряването на тази глава помогнаха д-р Стефан Вътев, Стефан Герджиков, Соломон Паси, проф. Димитър Скордев, Александра Соскова, проф. Тинко Тинчев и други. Нито един от тях не може по никакъв начин да се счита отговорен за допуснатите грешки. Също така, ни най-малко не може да се счита, че те споделят изцяло или дори част от изказаните в този текст нематематически тези.

Глава 2

Синтаксис

Фразите на един формален език всъщност не са редици от символи, а абстрактни обекти, означавани с редици от символи, също както естествените числа са абстрактни обекти, означавани с редици от цифри. Следователно да дефинираме семантиките посредством функции върху редици от символи би било също толкова заобиколно, колкото ако дефинирахме аритметичните функции върху редици от цифри.

Джон Рейнолдс *[15]*

2.1. Формални езици

Синтаксис и семантика

Правилата и принципите, въз основа на които може да формулираме правилни изрази от даден език, се нарича *синтаксис* на този език. Смисълът пък на така построените правилни изрази е *семантиката* този език.

Обичайният език, на който говорят хората е двусмислен и неточен. Затова, когато математиците формулират някоя дефиниция или твърдение, те спазват определени от математическата традиция правила. Благодарение на тези неписани правила, езикът на математиката има ясен за всеки математик смисъл. Обаче има случаи, когато спазването

```
function НОД(m, n)
    if n = 0
       return m;
    else
       return НОД(n, m mod n);
```

Фиг. 1. Примерен псевдокод за алгоритъма на Евклид

```
function НОД(M, N: Natural) return Natural is
begin
   if N = 0 then
      return M;
   else
      return НОД(N, M mod N);
   end if;
end НОД;
```

Фиг. 2. Програма на ада за алгоритъма на Евклид

неписани правила е недостатъчно. В тези случаи използваме т. н. формални езици. *Формални езици* са онези езици, чиито синтаксис и семантика са формулирани точно и недвусмислено.

За да бъде един език формален, не е достатъчно той да има ясен и точен смисъл. Например псевдокодът, който в книгите по програмиране се използва за описание на различни алгоритми, е пример за език с точен смисъл, който обаче не е формален език. Макар и да е разбираем за читателя, псевдокодът, няма формално дефинирани синтаксис и семантика. Езиците за програмиране пък са примери за формални езици.

Сравнете програмите от фиг. 1 и фиг. 2. Те представят рекурсивния вариант на алгоритъма на Евклид съответно на псевдокод (точен математически език) и на ада (формален език). Няма да се случи нищо нередно, ако в края на първия ред на програмата от фиг. 1 добавим двоеточие, защото смисълът на така променения програмен текст остава напълно ясен. Обаче в програмата от фиг. 2 не можем да правим подобни промени, защото езикът ада си има точно определен синтаксис, който не позволява да слагаме двоеточия където поискаме.

За да се направят удобни за използване от хора, компютърните формални езици често имат много сложен синтаксис. За да се опише син-

таксисът на някой език за програмиране, често са нужни стотици синтактични правила, определящи неща като приоритетите на операциите, къде може и къде не може да слагаме интервали и празни редове и т. н. Това е много неудобно, когато искаме да обсъждаме семантиката на формалния език, защото се налага да си отвличаме вниманието с много ненужни синтактични подробности. Например когато се интересуваме от семантиката на един аритметичен израз, няма никакво значение дали сме го написали с интервали `x + y` или без интервали `x+y`.

Това е причината когато работим с формални езици да се използват два синтаксиса: *конкретен синтаксис* и *абстрактен синтаксис*. Конкретният синтаксис е сложен. Него използваме, когато пишем изрази от съответния език. Например една компютърна програма е написана съгласно правилата на конкретния синтаксис на използвания език за програмиране. Абстрактния синтаксис пък е максимално опростен. Него използваме тогава, когато искаме да разсъждаваме за смисъла на изразите от даден език, например как точно ще работи дадена програма. Въпреки че конкретният синтаксис е сложен, а абстрактният синтаксис е опростен, разликите между двата синтаксиса са несъществени от гледна точка на смисъла на изразите на формалния език.

Езици за хора и езици за компютри

Формалните езици се разделят на две големи групи — езици, които са предназначени да бъдат обработвани от компютър, и езици, при които компютърната обработка не е сред основните им цели.

Езиците за програмиране са примери за формални езици от първата група. Други примери за такива формални езици са езиците за заявки към бази данни, езиците XML и HTML, използвани за уеб-страниците в Интернет, и много други. Понякога да формулираме точния синтаксис и семантика на езиците от тази група по начин, разбираем за хората, е непосилна задача. Например до момента никой не е успял да формулира по разбираем начин какъв точно е синтаксисът на езика за програмиране пърл.*

*Езикът пърл май е единственият популярен език за програмиране с толкова сложен синтаксис. Синтаксисът на повечето езици за програмиране може да се опише по разбираем за хората начин. Но дори един език за програмиране да има прост синтаксис, неговата семантика най-често се оставя без точна математическа дефиниция. В някои случаи тази неопределеност е съществена. На програмистите им се налага да пишат примерни програми, за да видят как точно ще се изпълни даден код. Разбира се, няма гаранции, че следващите версии на компилатора ще изпълнят този код по същия начин.

Обаче без значение дали конкретният синтаксис на даден език за програмиране има разбираема дефиниция, този синтаксис винаги е фиксиран и точно определен. Не може да направим компилатор на даден език за програмиране, без да сме определили кои редици от символи ще считаме, че са правилни програми от съответния език. Що се касае до абстрактния синтаксис на компютърните формални езици, то такъв много често изобщо не се дефинира официално. Това е така, защото от абстрактния синтаксис имаме нужда само тогава, когато решим да разсъждаваме математически за семантиката на формалния език.

Втората група формални езици са тези, при които компютърната обработка не е сред основните им цели. Такива са например някои езици за спецификация на програми, както и разнообразните езици, използвани в математическата логика. Докато компютърните формални езици имат „официално“ дефиниран конкретен синтаксис, а абстрактния съществува неофициално, при „некомпютърните“ формални езици положението е обратното — те имат официално дефиниран абстрактен синтаксис, докато конкретният им синтаксис обикновено не се дефинира точно.

2.1.1. Пример. Да разгледаме език, чийто абстрактен синтаксис се задава от безконтекстна граматика с терминални символи `a`, `b`, `1`, `2`, `(`, `)`, `+` и `*`, начален символ S, единствен нетерминален символ S, и правила

$$S \longrightarrow \texttt{(}S\texttt{+}S\texttt{)} \mid \texttt{(}S\texttt{*}S\texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{1} \mid \texttt{2}$$

Ако използваме формата на Бекус – Наур, можем да запишем тези правила и така:

$$\langle S\rangle\texttt{::=(}\langle S\rangle\texttt{+}\langle S\rangle\texttt{)} \mid \texttt{(}\langle S\rangle\texttt{*}\langle S\rangle\texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{1} \mid \texttt{2}$$

Един примерен, правилно построен израз от този език е `(a+(2*b))`. Да забележим, че граматиката на този език ни задължава да слагаме скоби около всяка аритметична операция. Благодарение на това не се налага да се усложняваме с приоритети на операциите, неща като лява и дясна асоциативност на операциите и др. Няма никакъв проблем да се уговорим когато използваме този език да изпускаме подразбиращите се скоби. По този начин за удобство вместо `(a+(2*b))` може да пишем просто `a+2*b`. Важно е да знаем обаче, че работим с формален език, имащ точно определен синтаксис, в който аритметичните операции задължително се ограждат със скоби. Това означава, че дори когато сме написали `a+2*b`, ние всъщност имаме предвид `(a+(2*b))`. Например ако в някое математическо разсъждение се налага да преброим броя на символите в израза `a+2*b`, този брой ще бъде 9, а не 5.

В горния пример може да считаме, че дадената граматика дефинира точно абстрактният синтаксис на формалния език. Изрази като `a+2*b` пък може да считаме че отговарят на конкретния синтаксис на езика, който няма нужда да дефинираме точно, защото езикът не е предназначен за компютри, а за хора.

Има много формални езици, за които не се е предвиждало да се обработват с компютър, но в последствие това се е оказало полезно. В такива случаи компютърната реализация на формалния език използва по-удобен за използване синтаксис в сравнение със синтаксиса от първоначалната математическа дефиниция. С други думи, преди да се появи компютърната реализация, конкретният синтаксис на езика е оставал неформално дефиниран, но след появата на компютърна реализация този синтаксис е станал точно определен и описан в софтуерната документация.

Забележка: Може да направим следните изводи:

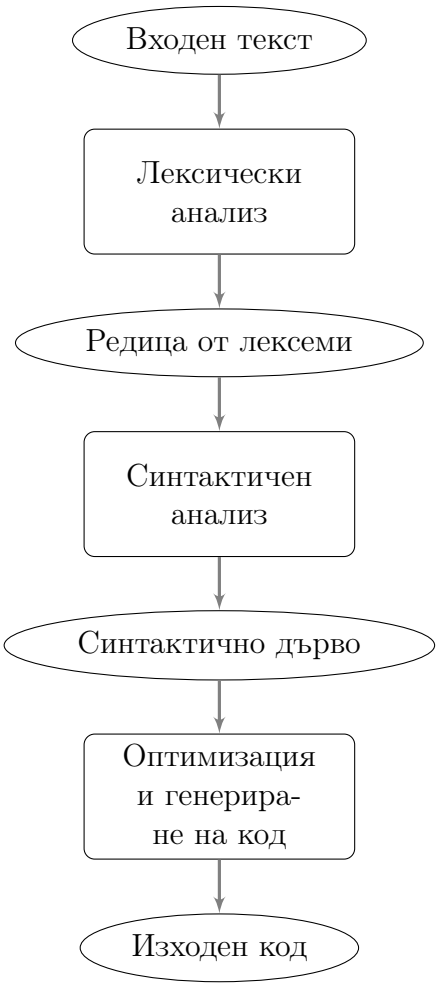
- Ако даден формален език е предназначен за компютърна обработка, той със сигурност има точно определен конкретен синтаксис.
- Ако даден формален език не е предназначен за компютърна обработка, той може и да няма точно определен конкретен синтаксис. Например може по напълно неформален начин „да се уговорим“ да изпускаме някои от скобите или да добавяме интервали за по-добра четливост.
- Ако искаме да правим точни математически разсъждения за изразите от даден формален език, то имаме нужда от точно определен абстрактен синтаксис.

Синтактични дървета

Въпреки че езиците за програмиране нямат официално дефиниран абстрактен синтаксис, оказва се, че когато пишем компилатор на даден език, има нужда да дефинираме точно абстрактния синтаксис на езика. Това не е изненадващо, тъй като точната интерпретация на програмата е свързана с математически преобразувания, на които се основава коректността на компилатора.

За да се научим да дефинираме по хубав начин абстрактния синтаксис на даден формален език, напр. език за програмиране, нека видим как всъщност работят транслаторите на езиците за програмиране.

Транслаторът е *програма*, която преобразува програма от един формален компютърен език (т. н. *входен език*) на друг формален компютърен език (т. н. *изходен език*). Когато входният език е от високо



Фиг. 3. Обичаен строеж на транслаторите

ниво (напр. ада, си, джава), а изходният — от ниско (машинен език или байт-код), транслаторът се нарича *компилатор*. Когато пък входният език е от ниско ниво, а изходният — машинен език, тогава транслаторът се нарича *асемблер*. Има и транслатори, които превеждат от един език от високо ниво на друг език от високо ниво. Такива транслатори се наричат *конвертори*.

Обикновено транслацията се извършва на няколко стъпки, както това е показано във фигура 3.

Първата стъпка от процеса на транслацията се извършва от т. н. лексически анализатор. На входа му постъпва програмният текст под формата на редица от символи (напр. илюстрирания във фигура 2). След като бъде обработена от лексическия анализатор, програмата се превръ-

function | нод | (| m | , | n | : | natural |) | return | natural | is | begin | if
n | = | 0 | then | return | m | ; | else | return | нод | (| n | , | m | mod | n
) | ; | end | if | ; | end | нод | ;

Фиг. 4. Редица от лексеми за програмата на ада от фиг. 2

ща в редица от лексеми. Примерна редица от лексеми, отговаряща на програмата от фиг. 2, е показана във фигура 4.* Както може да се забележи, всяка лексема представлява или дума (напр. идентификатора `Natural` и служебната дума `return`), или литерал (число, низ, и др.), или разделител, или оператор, или коментар. Като основна техника за реализиране на лексическия анализатор се използват крайни автомати. Благодарение на това алгоритмите, използвани от лексическия анализатор са много бързи.**

След като от лексическия анализ получим редица от лексеми, преминаваме към втората стъпка от транслацията — синтактичния анализ. Целта на синтактичния анализатор е да се получи т. н. синтактично дърво. Като основна техника за реализиране на синтактичния анализатор се използват безконтекстни граматики от специален вид. ***

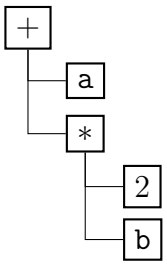
Смисълът на синтактичното дърво е да изрази ясно структурата на израза във вид, удобен за последваща комютърна обработка. Да разгледаме например аритметичния израз „$\mathtt{a}+2*\mathtt{b}$“. Никак не е очевидно как

*В езика ада не се прави разлика между малки и главни букви. Например „Natural“, „natural“ и „NATURAL“ са един и същи идентификатор. Това е причината, поради която в лексемите от фигура 4 са използвани само малки букви.

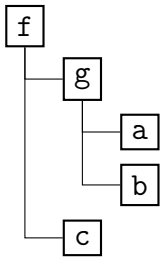
**По принцип цялата работа на лексическия анализатор може да се свърши и от синтактичния. Ако не се интересувахме от бързината на компилатора, то изобщо не би имало нужда от лексически анализатор.

***Алгоритмите, които могат да се използват за синтактичен разбор на произволна безконтекстна граматика са неефективни. Например алгоритъмът на Кок – Янгър – Касами за разбор на безконтекстна граматика, приведена в нормална форма на Чомски, който обикновено се учи в курсовете по дискретна математика, дискретни структури и езици, автомати и изчислимост във ФМИ, е с кубична сложност. Безконтекстните граматики на повечето езици за програмиране обаче имат специални свойства, които позволяват разборът да се извършва за линейно време (макар че дори и в този случай синтактичният анализ е значително по-бавен от лексическия).

Както е известно, крайните автомати разпознават по-тесен клас езици от безконтекстните граматики. Поради това не е възможно да елиминираме нуждата от безконтекстни граматики и да разчитаме изцяло на бързите алгоритми от лексическия анализатор. Форт е един извънредно рядък пример за език от високо ниво, който обаче има регулярна граматика и поради това може да се анализира без да се използват безконтекстни граматики.



Фиг. 5. Синтактично дърво на $\texttt{a} + 2 * \texttt{b}$



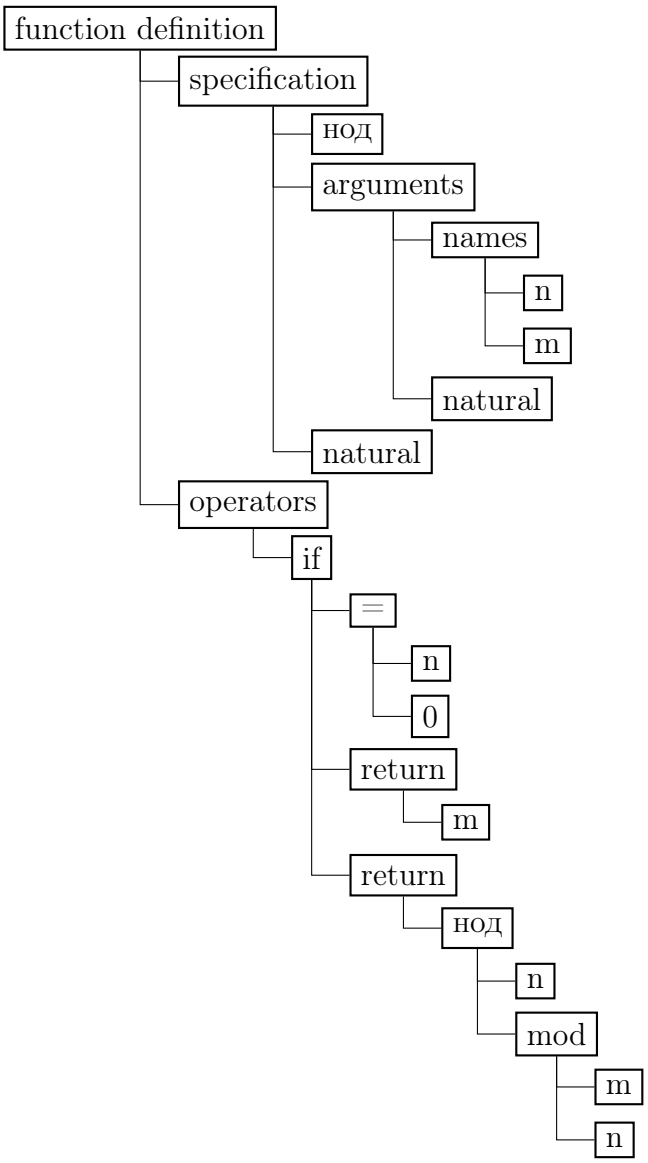
Фиг. 6. Синтактично дърво за израза $\texttt{f}(\texttt{g}(\texttt{a}, \texttt{b}), \texttt{c})$

може да се напише компютърна програма, която пресмята стойността на такъв израз. Как например тази програма ще определи с какво трябва да съберем `a` — с 2 или с $2 * \texttt{b}$? Само от израза не можем да отговорим на този въпрос, защото от него не личи какъв е приоритетът на различните операции. Нека разгледаме обаче съответното синтактично дърво от фигура 5. Това дърво не оставя никакви съмнения, че `a` се събира с $2 * \texttt{b}$, а рекурсивният му вид прави алгоритмичното пресмятане на аритметичния израз сравнително проста задача.

Синтактичното дърво на израза $\texttt{f}(\texttt{g}(\texttt{a}, \texttt{b}), \texttt{c})$ е илюстрирано във фигура 6. Във фигура 7 пък е показано примерно синтактично дърво за програмата от фигура 2.

***Задача* 1:** Напишете синтактични дървета за следните изрази на езика си: `x+(y+z)` и `(x+y)+z`. Обърнете внимание, че в си операцията `+` е лявоасоциативна, така че изразите `(x+y)+z` и `x+y+z` имат едно и също синтактично дърво. Същото синтактично дърво имат и изразите `((x+y))+z` и `(x)+y+z`.

***Задача* 2:** Напишете синтактични дървета за следните изрази на езика си: `x+(++y)`, `(x++)+y` и `x+++y`.



Фиг. 7. Примерно синтактично дърво за програмата от фиг. 2

Подходи при дефиниране на абстрактния синтаксис

Вече видяхме, че компилаторите преобразуват първоначалния текст на програмата в синтактични дървета. Обработката на синтактичните дървета е много по-лесна и естествена, отколкото ако компилаторът трябваше да работи директно върху първоначалния програмен текст. Синтактичните дървета не само имат по-ясен смисъл, който не зависи например от приоритетите на операциите, но освен това ни спестяват и други ненужни подробности. Например изрази като `a+2*b` и `a+  ((2) *b)`, които очевидно казват едно и също нещо, ще получат едно и също синтактично дърво — дървото от фигура 5.

Същото обаче важи и за математическите разсъждения. Когато се интересуваме от смисъла на формалните изрази, много по-лесно ще бъде да разсъждаваме върху синтактичните им дървета, отколкото върху редици от символи. Това ни дава и ключа за правилното дефиниране на абстрактния синтаксис на даден формален език. Абстрактният синтаксис трябва така да бъде дефиниран, че връзката между изразите при този синтаксис и съответните синтактични дървета да бъде възможно най-естествена и непосредствена.

Някои математици, особено работещите в областта на компютърните науки, приемат, че обектите от абстрактния синтаксис на даден формален език са не редици от символи, а самите синтактични дървета. Ясно е, че по този начин връзката между обектите отговарящи на абстрактния синтаксис и синтактичните дървета е възможно най-непосредствена. По-традиционните математици пък по-често използват абстрактен синтаксис, в който изразите все пак са редици от символи, а не дървета. От математическа гледна точка двата подхода са напълно еквивалентни.

В този курс ще използваме „традиционния“ подход, т.е. изразите от даден формален език ще бъдат редици от символи, а не дървета. Това има следните предимства. Първо, тъй като подходът, при който формалните изрази всъщност са дървета е по-абстрактен, той създава трудности, напр. у студенти с незатвърдени математически навици. В даден момент може да забравим, че редицата от символи, която сме написали, всъщност е някакво дърво, а не това, което изглежда, че е. Второ, ако изразите са дървета, ще трябва да обърнем по-голямо внимание на конкретния синтаксис, защото все пак трябва да имаме някакъв начин да записваме изразите като редици от символи. При „традиционния“ подход след като дефинираме какъв е абстрактния синтаксис, обикновено става ясно какъв е и конкретния и няма нужда да му обръщаме голямо внимание. Разбира се, посочените тук две предимства не

са решаващи, така че кой какъв подход ще използва, зависи най-вече от личните предпочитания.

Ще разгледаме някои техники, които често се използват, за да може дефиницията на абстрактния синтаксис на даден език да бъде проста за използване, а съответствието между изразите от езика и синтактичните им дървета — очевидно.

Математически идеализации

Математическите обекти винаги представляват идеализации, които притежават само онези свойства, които са нужни на математиците. Например точките в геометрията имат точно определено местоположение, но нямат нито дължина, нито ширина, нито височина, нито ориентация. Разбира се, в реалния свят няма обекти без размер. Въпреки това, има много ситуации, при които се интересуваме единствено от местоположението на дадени обекти, но не и от техните размери или ориентация. В тези случаи е удобно да считаме, че тези обекти са точки.

Нека забележим, че е погрешно да считаме, че размерите на обектите, отъждествявани с точки, са пренебрежимо малки. Например в определени ситуации един астроном може да счита, че планетите и звездите са точки в пространството. А в други ситуации не можем да приемем за точкови дори частици като протоните и неутроните.[*] Това, че считаме дадени обекти за точки, няма никакво отношение към размерите на тези обекти, а означава само едно — че не се интересуваме от техните размери и ориентация, а само от местоположението им.

Всъщност оказва се, че геометрията може да съществува и без в нея да се използват точки.[**] При това се оказва, че в безточковите геометрии можем да правим съвсем същите неща, които и в обикновените геометрии. При безточковите геометрии обаче се работи по-сложно, защото когато искаме да обозначаваме позиции в пространството не можем за целта да използваме просто точки, а се налага да правим това по заобиколен начин.[***]

[*]Например при експерименти с дълбочинно нееластично разсейване се проявяват трите кварка, от които са образувани тези частици.

[**]Теорията на безточковите геометрии е разработвана и от много български математици — Николай Белухов, проф. Димитър Вакарелов, проф. Георги Димов, Татяна Иванова, Владислав Ненчев, проф. Тинко Тинчев.

[***]Квантовата механика ни кара да се запитаме има ли всъщност точки в пространството и не е ли всъщност геометрията на реалния свят безточкова. Засега обаче никой не е разработил безточкова геометрия, която да съответства на света квантовата механика. Всички разработени до момента безточкови геометрии са по един или друг начин еквивалентни на обикновената евклидова геометрия. Регионите, с

Компютърните програми използва ключови думи и идентификатори като `return`, `while`, `вмъкване`, `helper4` и `helper_of_helper4`.* Да си припомним, че лексическият анализатор превръща тези думи в *лексеми*, след което синтактичният анализатор работи с тези лексеми все едно, че те се състоят от един единствен символ. Ясно е, че това е коректно и не води до неправилна работа на компилатора, защото в нито един случай вътрешната структура на тези имена не е от значение за семантиката на програмата. Например не е от значение, че думата `return` се състои от шест символа, а не от пет. Също така без никакво значение е и това, че името `helper_of_helper4` е съставено от три английски думи, а `вмъкване` — от една българска.

Това не е някакво изключително свойство на езиците за програмиране, а важи и за останалите формални езици, които ще използваме. Затова в математическите дефиниции на формалните езици, които ще използваме, ще приемаме, че всяко едно име в тях се състои от един единствен символ. Разбира се, в действителност запазената дума `return` не се състои от един символ, а от шест. Това обаче за нашите цели ще бъде без значение. Също и планетите не са точки, но понякога астрономът може да си мисли, че са.

Когато реално пишем изрази от даден формален език, не пречи да използваме многосимволни имена като `return` и `вмъкване`. Ще считаме, че конкретният синтаксис на езика позволява използването на такива имена. Само на ниво абстрактен синтаксис тези имена „ще се превръщат“ в символи

Претоварване

В много програмни езици имената на функции, методи, процедури и т. н. могат да се претоварват. Това означава, че на две напълно различни функции могат да се дадат едни и същи имена, стига начи-

---

които се работи в тези геометрии, имат точно определени граници, докато обектите в квантовата механика не притежават точно определени размери, координати и скорост. Тази неопределеност не се дължи на ограничените ни познания, а е фундаментална характеристика на света. Съгласно една теорема на Джон Стюард Бел, ако допълним квантовата механика с допълнителни параметри, които да направят нещата в нея точно определени, то получената теория или няма да дава същите предсказания за поведението на света, както съществуващата квантова механика, или ще трябва да допуска предаване на информация със скорост по-бърза от скоростта на светлината. Това е така, дори ако предположим, че тези допълнителни параметри са „скрити“ и недостъпни за наблюдение.

*Умолявам ви, не използвайте думата `helper` в програмите на пролог, когато се явявате на изпит по логическо програмиране!

нът, по който функциите се извикват, да ни позволява да различаваме коя точно функция се има предвид. Например на пролог едноименни предикати с различен брой аргументи се считат за напълно различни предикати. На си++ не само броят на аргументите на една функция или метод е от значение, но също и типовете на аргументите. А при ада дори типът на връщаната стойност е от значение. Впрочем претоварването не винаги е желателно. Затова някои езици за програмиране умишлено го забраняват. Например в езикът SPARK който представлява подмножество на ада и се използва за писане на програми без грешки, претоварването е забранено.*

Но дори когато претоварването е удобно, то всъщност не добавя никакви съществено нови възможности към езика за програмиране. Ако в даден език за програмиране претоварването е забранено, просто можем да даваме различни имена на различните функции, методи и т. н. Това наблюдение важи не само за езиците за програмиране, но и за останалите формални езици, които ще използваме. Вместо да пишем изрази като $\mathtt{f}(\mathtt{a},\mathtt{b})$ и $\mathtt{f}(\mathtt{a},\mathtt{b},\mathtt{c})$, винаги можем вместо това да пишем напр. $\mathtt{f}_2(\mathtt{a},\mathtt{b})$ и $\mathtt{f}_3(\mathtt{a},\mathtt{b},\mathtt{c})$. С други думи, вместо да използваме символа $\mathtt{f}$ едновременно като двуаргументен и триаргументен, може да използваме два различни символа $\mathtt{f}_2$ и $\mathtt{f}_3$. И тъй като това ще ни облекчи когато правим математически разсъждения, ние ще постъпим точно така — ще приемем, че символите във формалните езици имат еднозначно определен брой аргументи.

Вече споменахме, че ограничението всяко име да се състои точно от един символ, на практика не пречи, защото в конкретния синтаксис на езиците нямаме такова ограничение. Също така, няма проблем в конкретния синтаксис да използваме и претоварване. Само на ниво абстрактен синтаксис претоварените символи „ще се превръщат“ в различни символи. Например когато пишем изрази като $\mathtt{f}(\mathtt{a},\mathtt{b})$ и $\mathtt{f}(\mathtt{a},\mathtt{b},\mathtt{c})$, може да считаме, че всъщност сме написали $\mathtt{f}_2(\mathtt{a},\mathtt{b})$ и $\mathtt{f}_3(\mathtt{a},\mathtt{b},\mathtt{c})$.

*Да — има технологии, позволяващи писането на програми (почти) без грешки. В някои случаи писането на програми без грешки просто е необходимост. Например и най-малката грешка в компютърна програма, управляваща пътнически самолет, може да се окаже фатална. На много места има метро, в което влаковете не се управляват от човек, а автоматично от компютър — тук също не искаме управляващата програма да има грешки. Имало е случаи на хора, които са били убити от медицински апарат, облъчил ги с фатална доза радиация поради софтуерна грешка. Една грешка в програма за електронен подпис може да вкара невинен човек в затвора. Компютризацията в обществото непрекъснато расте, а с това се увеличава и броят на приложенията, при които е нужно програмите да бъдат без грешки.

Сортове, типове, променливи и операционни символи

Следващата дефиниция съдържа важни уговорки за символите. Тези уговорки ще бъдат изпълнени до края на този курс.

2.1.2. Дефиниция. а) Нека ни е дадено множество, чиито елементи ще наричаме *сортове*. Изразите от вида $\langle t_1, t_2, \ldots, t_n \rangle \mapsto t$, където $t_1, t_2, \ldots, t_n, t$ са сортове и n може да бъде 0, ще наричаме *типове*.* Типовете са аналог на функционалните типове в езиците за програмиране, а за сортовете може да си мислим като за някакви примитивни типове.

б) За всеки от сортовете нека ни е дадено безкрайно множество от символи, които ще наричаме *променливи*. Всяка променлива си има точно определен и единствен сорт.

в) Нека са ни дадени и други символи, наречени *операционни символи*, като всеки операционен символ има точно определен и единствен тип.

г) Ако даден операционен символ е от тип $\langle t_1, t_2, \ldots, t_n \rangle \mapsto t$, ще казваме, че $\langle t_1, t_2, \ldots, t_n \rangle$ е неговата *арност*.

д) За символите $\forall$, $\exists$, λ, запетаята и скобите ще считаме че са запазени, т.е. те не са нито променливи, нито операционни символи.

Ако даден операционен символ е от тип $\langle t_1, t_2, \ldots, t_n \rangle \mapsto t$, то интуитивно може да си мислим, че $\langle t_1, t_2, \ldots, t_n \rangle$ са сортовете на аргументите му, а t — сортът на връщаната стойност.

В предния подраздел се уговорихме, че всеки операционен символ има точно определен брой аргументи. А тук виждаме, че е фиксиран не само броят на аргументите, но и сортовете им. Ще считаме, че това ограничение съществува само в абстрактния синтаксис. Когато реално пишем изрази от даден формален език (с други думи, когато използваме конкретния синтаксис), това ограничение отпада и може да претоварваме символите.

Индуктивни дефиниции и безконтекстни граматики

Ще разгледаме някои от по-често използваните начини за дефиниране на абстрактния синтаксис. Удобно е да направим това, използвайки конкретни примери. Затова нека **int** и **bool** са сортове, нека `x`, `y` и `z` са променливи от сорт **int** и нека още са ни дадени следните операционни символи:

*На английски: ranks.

`1` и `2` от тип $\langle\rangle \mapsto \mathbf{int}$;

`+` и `*` от тип $\langle \mathbf{int}, \mathbf{int} \rangle \mapsto \mathbf{int}$;

`=` и `<` от тип $\langle \mathbf{int}, \mathbf{int} \rangle \mapsto \mathbf{bool}$;

`?` от тип $\langle \mathbf{int} \rangle \mapsto \mathbf{bool}$.

`!` от тип $\langle \mathbf{bool} \rangle \mapsto \mathbf{bool}$.

`&` от тип $\langle \mathbf{bool}, \mathbf{bool} \rangle \mapsto \mathbf{bool}$.

В математическата логика за дефиниране на синтаксиса на различни формални езици най-често се използват индуктивни дефиниции. Например изрази от сорт **int** може да дефинираме по следния начин:

2.1.3. Дефиниция. Ще дефинираме фразичките индуктивно:

а) Ако `x` е променлива от сорт **int**, то `x` е фразичка;

б) `1` и `2` са фразички;

в) Ако α_1 и α_2 са фразички, то низовете `(`α_1`+`α_2`)` и `(`α_1`*`α_2`)` са фразички.

г) Ако α е фразичка, то това може да се установи посредством краен брой прилагания на горните три правила.

Когато даваме индуктивна дефиниция, винаги се подразбира изречение, подобно на написаното по-горе с дребен шрифт. Затова, когато математиците дават индуктивни дефиниции, те обикновено изпускат това изречение.

Да дефинираме и изрази от сорт **bool**.

2.1.4. Дефиниция. Ще дефинираме фразищата индуктивно:

а) Ако α_1 и α_2 са фразички, то изразите `(`α_1`=`α_2`)` и `(`α_1`<`α_2`)` са фразища.

б) Ако α е фразичка, то низът α`?` е фразище.

в) Ако β е фразище, то низът `!`β е фразище.

г) Ако β_1 и β_2 са фразища, то низът `(`β_1`&`β_2`)` е фразище.

д) Ако β е фразище, то това може да се установи посредством краен брой прилагания на горните четири правила.

При по-сложни формални езици използването на словесни индуктивни дефиниции става непрактично. Затова такива езици обикновено се дефинират с помощта на безконтекстни граматики. За всеки сорт в граматиката ще има съответен нетерминален символ. Например нека

I е нетерминалният символ на сорта **int**, а B е нетерминалният символ на сорта **bool**. Нека освен това V е нетерминален символ, който ще разпознава променливите от сорт **int**. Да разгледаме следните правила:

$$I \longrightarrow V \mid \texttt{1} \mid \texttt{2} \mid \texttt{(}I\texttt{+}I\texttt{)} \mid \texttt{(}I\texttt{*}I\texttt{)}$$
$$B \longrightarrow \texttt{(}I\texttt{=}I\texttt{)} \mid \texttt{(}I\texttt{<}I\texttt{)} \mid I\texttt{?} \mid \texttt{!}B \mid \texttt{(}B\texttt{\&}B\texttt{)}$$

В такъв случай ако направим I да бъде начален символ на граматиката, тогава езикът ѝ ще бъде множеството от всички фразички. Ако пък направим B да бъде началният символ, то тогава езикът ще бъде множеството от всички фразища.

Горните правила записахме по начина, който най-често се използва от математиците. А хората, занимаващи се с компютърни приложения, често използват т. н. форма на Бекус – Наур. При нея заграждаме нетерминалните символи в ъглови скоби, вместо букви като V, I и B използваме смислени имена и вместо стрелка използваме знака ::= по следния начин:

⟨фразичка⟩ ::=⟨променлива⟩ | 1 | 2
| (⟨фразичка⟩ + ⟨фразичка⟩)
| (⟨фразичка⟩ * ⟨фразичка⟩)
⟨фразище⟩ ::=(⟨фразичка⟩ = ⟨фразичка⟩)
| (⟨фразичка⟩ < ⟨фразичка⟩)
| ⟨фразичка⟩?
| !⟨фразище⟩
| (⟨фразище⟩ & ⟨фразище⟩)

Структурна индукция

Винаги, когато даваме индуктивна дефиниция на дадено понятие, е в сила съответен принцип за индукция. Сравнете внимателно трите правила от дефиницията на фразичките с трите условия в следния принцип за индукция:

2.1.5. Принцип за индукция по построението на фразичките. *Нека* $\mathfrak{p}$ *е такова свойство, че:*

а) *ако* x *е променлива от сорт* **int**, *то* x *притежава свойството* $\mathfrak{p}$;

б) *едносимволните низове* 1 *и* 2 *притежават свойството* $\mathfrak{p}$;

в) *ако* α_1 *и* α_2 *са фразички, които притежават свойството* $\mathfrak{p}$, *то също и низовете* $(\alpha_1\texttt{+}\alpha_2)$ *и* $(\alpha_1\texttt{*}\alpha_2)$ *притежават свойството* $\mathfrak{p}$.

В такъв случай всички фразички притежават свойството $\mathfrak{p}$.

Доказателство. Нека α е произволна фразичка.

Тъй като α е фразичка, то това може да бъде доказано, използвайки краен брой пъти трите правила от дефиниция 2.1.3. В едно такова доказателство ще се формулират твърдения, че определени изрази са фразички. Нека $\sigma_1, \sigma_2, \ldots, \sigma_m$ са всички изрази, за които в това доказателство се казва, че са фразички, подредени според срещането им в доказателството. Ще докажем, че фразичките $\sigma_1, \sigma_2, \ldots, \sigma_m$ притежават свойството $\mathfrak{p}$. Тъй като в това доказателство със сигурност ще е изказано и твърдението, че α е фразичка (т.е. $\alpha = \sigma_i$ за някое i), то по този начин ще получим исканото.

Да изберем $i \in \{1, 2, \ldots, m\}$. Ще докажем, че σ_i притежава свойството $\mathfrak{p}$ с пълна математическа индукция по i. Индукционното предположение ще казва, че фразичките $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ притежават свойството $\mathfrak{p}$.

Тъй като в разглежданото доказателство се позволява да се използват единствено трите правила от дефиниция 2.1.3, то значи фактът, че σ_i е фразичка, е доказан с някое от тези три правила.

Ако този факт е доказан с първото правило, то значи σ_i е променлива от сорт **int**. В такъв случай от правило 2.1.5 а) получаваме, че σ_i притежава свойството $\mathfrak{p}$.

Ако този факт е доказан с правило 2.1.3 б), то значи σ_i е някой от символите `1` или `2`. В такъв случай от правило 2.1.5 б) получаваме, че σ_i притежава свойството $\mathfrak{p}$.

Остава възможността това да е било доказано с правило 2.1.3 в). В този случай σ_i има вида `(`α_1`+`α_2`)` или `(`α_1`*`α_2`)`, където α_1 и α_2 са изрази, за които по-рано в разглежданото доказателство вече е било доказано, че са фразички. Съгласно индукционното предположение $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ притежават свойството $\mathfrak{p}$, а значи и α_1 и α_2 притежават свойството $\mathfrak{p}$ и от правило 2.1.5 в) получаваме, че σ_i притежава свойството $\mathfrak{p}$. ■

Забележка: Много е важно този принцип за индукция да не се учи наизуст, а човек сам да може да съобразява какво гласи той, помнейки единствено каква е дефиницията на фразичка. Доказателството му трябва да се разбере, но също не е нужно да се учи.

2.1.6. Пример. Ще използваме принципа за индукция за фразички, за да докажем, че всички фразички съдържат равен брой леви и десни скоби.

Доказателство. а) Ако `x` е променлива, то `x` съдържа равен брой леви и десни скоби (нула), защото `x` е променлива, а за скобите се уговорихме, че са запазени символи и значи не се използват като символи за променливи; вж. дефиниция 2.1.2 д).

б) Символите `1` и `2` съдържат равен брой леви и десни скоби (нула), защото са операционни символи от тип $\langle\rangle \mapsto \mathbf{int}$, а за скобите се уговорихме че са запазени символи и значи не са операционни символи; вж. дефиниция 2.1.2 д).

в) Да допуснем, че α_1 и α_2 са фразички с равен брой леви и десни скоби.* Нека k_i е броят на левите скоби в α_i (същото число е равно и на броя на десните скоби). Тъй като символите `+` и `*` са операционни символи от тип $\langle \mathbf{int}, \mathbf{int}\rangle \mapsto \mathbf{int}$, то те не са скоби. Следователно броят на левите скоби във фразичките `(`α_1`+`α_2`)` и `(`α_1`*`α_2`)` е $1 + k_1 + k_2$ — една отворена скоба в началото плюс скобите в α_1 и α_2. Броят на десните скоби е същият — затворената скоба в края на фразичката плюс скобите в α_1 и α_2. ∎

***Задача* 3:** Да разгледаме следната индуктивна дефиниция на понятието „буртант“:

а) 5 е буртант;

б) 8 е буртант;

в) Ако n и m са буртанти, то nm^2 е буртант.

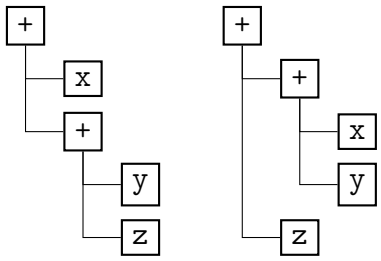
Какъв е индуктивният принцип за буртанти? Използвайки го, докажете, че ако n е буртант, то $n + 1$ е естествено число, което се дели на 3 без остатък.

Видове формални записи

2.1.7. Пример (инфиксен, префиксен и постфиксен запис). Да погледнем отново дефинициите на фразичките и фразищата. Може да забележим, че синтаксисът, който използвахме за операциите `+`, `*`, `=`, `<` и `&`, бе идентичен. Такъв начин за записване на операциите се нарича *инфиксен запис*. Символът `?` се поставя след фразичката, за която се отнася; такъв запис се нарича *постфиксен запис*. Символът `!` пък поставяхме преди фразището, за което се отнася; такъв запис се нарича *префиксен запис*.

Да забележим, че дефинициите на фразичките и фразищата ни задължават да слагаме скоби винаги когато използваме инфиксни операции. Например следните изрази са фразички: `1`, `(1+x)`, `(x+(2*y))`,

*Това допускане са нарича *индукционно предположение*.



Фиг. 8. Две синтактични дървета за израза `x+y+z`

обаче изразът `x+(2*y)` не е фразичка. Благодарение на това не се налага да се усложняваме с приоритети на операциите, неща като лява и дясна асоциативност и др. Ако дефинициите не ни задължаваха да слагаме скоби, тогава изразът `x+y+z` би се оказал фразичка с две различни синтактични дървета (вж. фигура 8).

Когато комбинираме инфиксни и префиксни операции, тогава няма нужда да слагаме скоби около префиксните. Също така, когато комбинираме инфиксни и постфиксни операции, няма нужда от скоби около постфиксните. Едновременното използване без скоби на префиксни и постфиксни операции обаче може да доведе до двусмислие. При фразичките и фразищата нямаме проблем със символите `?` и `!` само защото единия се прилага към фразички, но не и към фразища, а другия към фразища, но не и към фразички. Например във фразището `!x?` е ясно, че прилагаме `!` към `x?`, а не `?` към `!x` (защото `!x` не е фразичка, а `?` се прилага към фразички).

Въпреки че формалният инфиксен запис ни задължава да слагаме повече скоби, отколкото обикновено правим, той все пак е лесен за четене от хора. Разбира се, скобите са задължителни само в абстрактния синтаксис на езика. В конкретния синтаксис може да изпускаме ненужните скоби, както и да се уговорим какви ще бъдат приоритетите и асоциативностите на операциите.

2.1.8. Пример (полски запис). Както вече бе отбелязано, често при формалните езици е важно те да имат прост синтаксис, който да позволява лесно да доказваме свойствата на тези езици, дори когато точното спазване на този синтаксис е неудобно. През 1924 полският математик Ян Лукашевич измислил т. н. *полски запис*, при който изобщо не се налага да пишем скоби, не се налага да говорим за приоритет и асоциативност на операциите, а свойствата на формалните езици с полски запис, се доказват по-лесно, отколкото при използване на запис със скоби. При полския запис първо записваме операцията, а веднага след

нея, без скоби или каквито и да е други разделители, аргументите ѝ. Например изразът $x + y$ в полски запис изглежда така: $+xy$. Фразичката `(x+(2*y))` в полски запис изглежда така: `+x*2y`. Сравнете това с фразичката `((x+2)*y)`, чийто полски запис изглежда така: `*+x2y`.

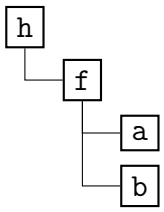
Въпреки че полският запис опростява формалните разсъждения, той има един съществен недостатък — обикновено хората не записват изразите по този начин. Затова в днешно време този запис се използва рядко. Вместо това се използват формални езици, при които се поставят всички скоби както в пример 2.1.7 и след това се уговаряме, че в конкретния синтаксис ще изпускаме скобите, когато няма опасност от недоразумения.

2.1.9. Пример (обратен полски запис). Ако записваме операции не преди, а след операндите, получаваме т. н. *обратен полски запис*. Фразичката `(x+(2*y))` в обратен полски запис става `x2y*+`, а фразичката `((x+2)*y)` става `x2+y*`. Обратният полски запис рядко се използва за формални езици, но за сметка на това има много приложения в информатиката. Обратният полски запис е открит първо от Артър Бъркс, Дан Уорън и Джеси Райт [1] през 1954 г., след което е бил преоткриван и от други математици и информатици.

2.1.10. Пример (традиционен функционален запис). Обикновено в математиката прилагането на някоя функция към аргументи се записва по следния начин: $f(a_1, a_2, \ldots, a_n)$. С други думи, най-напред записваме името на функцията, след това лява скоба, след това аргументите на функцията, разделени със запетая, и накрая дясна скоба. Този запис може да наречем *традиционен функционален запис*. Фразичката `(x+(2*y))` при традиционния функционален запис става `+(x,*(2,y))`, а фразичката `((x+2)*y)` става `*(+(x,2),y)`.

2.1.11. Пример (модерен функционален запис). Под влияние на λ-смятането и функционалните езици, в последно време някои математици за прилагането на функция към аргументи използват не традиционния запис $f(a_1, a_2, \ldots, a_n)$, а $(f a_1 a_2 \ldots a_n)$. Разликата между този запис и полския запис е това, че около функционалните извиквания се слагат скоби. Този запис може да наречем *модерен функционален запис*. Фразичката `(x+(2*y))` при модерния функционален запис става `(+x(*2y))`, а фразичката `((x+2)*y)` става `(*(+x2)y)`. В конкретния синтаксис най-външните скоби се изпускат.

Ако ни е дадено някое синтактично дърво, не е трудно да напишем израз в традиционен или модерен функционален запис, който има то-



Фиг. 9. Синтактично дърво за h(f(a, b)) и (h(fab))

ва синтактично дърво. Например на синтактичното дърво от фигура 6 отговарят изразите f(g(a, b), c) и f(gab)c, на синтактичното дърво от фигура 9 отговарят изразите h(f(a, b)) и h(fab), а на синтактичното дърво от фигура 5 отговарят изразите +(a, ∗(2, b)) и +a(∗2b). Следователно функционалният запис (също както и полският) е универсален.

Забележка: Ако от един израз, записан в традиционен или модерен функционален запис, отстраним всички скоби и запетаи, получаваме израз в полски запис. Например от `+(x,*(2,y))` и `(+x(*2y))` получаваме `+x*2y`.

2.2. Съждителна логика

Исторически бележки

Обществото на Древна Гърция е интересно явление. В него учените се считали за едни от най-видните членове на обществото. Всички се вълнували от откритията им. И най-обикновените хора развълнувано обсъждали споровете им по пазарите. „Жълтите“ журналисти разпространявали клюки какво се случило с този или онзи учен. Държавата ги увековечавала със скулптури.*

Именно в Древна Гърция — място, където научните изследвания се правели не за печалба, а от любов към знанието — се появили и първите логически изследвания. Безспорен основател на логиката като научна дисциплина е Аристотел. Той систематизирал изследванията на предшествениците си и създал т. н. *силогистика* — сравнително прост от съвременна гледна точка логически език. Аристотел изследвал какъв е

*След като Платон измислил утопична теория за това каква трябва да бъде идеалната държава, оказало се, че не било чак толкова трудно да намери малко царство, където да реализира на практика представите си за идеалната държава. Експериментът завършил зле както за Платон, така и за злополучния местен цар — и двамата били изгонени позорно от разбунтуваните граждани.

точният смисъл на съжденията, които можем да изкажем, използвайки езика на силогистиката, както и начините за доказателство на такива съждения. По времето на Аристотел обаче все още никой не бил изследвал точния смисъл на обичайните съждителни логически операции като „и“, „или“, „ако . . . , то“.

Оказало се, че импликацията е една от най-проблемните логически операции. Когато в обикновен разговор направим изявлението „ако вали дъжд, то времето е облачно“, с това ние казваме, че има някаква причинно-следствена връзка между валенето на дъжд и облачното време. Никога в обикновен разговор няма да чуем твърдение от вида „ако Витоша е планина, то София е град“ или „ако репичката е синя, то Пенчо обича Марийка“, защото не се вижда да има някаква връзка между това Витоша да бъде планина, а София — град, нито между цвета на репичката и любовта на Пенчо. Оказва се обаче, че ако искаме да разсъждаваме правилно, се налага да влагаме в импликацията по-различен смисъл, отколкото при ежедневното общуване. Ако съждението B е вярно, то и импликацията „ако A, то B“ е вярна без значение какво е съждението A. Тази импликация е вярна също и тогава, когато е невярно съждението A, без значение какво е съждението B.

Това че при обичаен разговор ние използваме един вид импликация, а при точни разсъждения — друг, не било забелязано веднага и отначало водело до грешки. Например Аристотел неправилно считал, че от едно съждение A никога не може да следва неговото отрицание $\neg A$. Главна заслуга за разкриване на смисъла на импликацията има т. н. Мегарска школа. Именно в тази школа Диодор Крон е дал първата широко възприета в античността дефиниция на смисъла на импликацията. А неговият ученик Филон Мегарски е дал дефиниция, която по същество съвпада с дефиницията, която използваме в съвременната съждителна логика.

Въз основа на всички тези изследвания, около сто години след Филон, в една друга школа — тази на стоиците — Хризип създал съждителната логика.* Хризип е написал над 700 научни съчинения, като почти половината от тях са посветени на логиката. Заслугите му били общопризнати в античността. Например според древния биограф Диоген Лаертски „ако боговете използваха логика, то те щяха да използват не някоя друга, а логиката на Хризип“. А през втори век християнският писател Климент Александрийски го нарича „майсторът на логиците,

*Всъщност Хризип създал нещо повече от съждителната логика — логиката на Хризип е безкванторният фрагмент на предикатната логика без функционални символи.

също както Омир е майсторът на поетите“. За съжаление през средновековието постиженията на Хризип не се разбирали и били забравени. От всичките му логически съчинения до наши дни не е оцеляло нищо, така че за тях се налага да съдим главно въз основа на цитати у по-късни автори.

Изречения, изявления, съждения, твърдения

Когато говорим използваме *изречения*. За някои изречения е безсмислено да казваме, че са верни или неверни, например „О, мило мое логическо програмиране!“. Изреченията, за които това не е безсмислено, се наричат *изявления*. Ето пример за изявления на четири различни езици (български, английски, полски и сръбски):

> Импликацията е невярна, ако предпоставката ѝ е вярна, а заключението ѝ — невярно; във всички други случаи импликацията е вярна.
>
> The implication is false if its antecedent is true and its consequent is false; in all other cases the implication is true.
>
> Implikacja jest więc fałszywa, jeśli jej poprzednik jest prawdziwy, a następnik fałszywy; we wszystkich innych przypadkach implikacja jest prawdziwa.
>
> Импликација је лажна ако њен антецедент је истина, а њен консеквент је лажан; у свим осталим случајевима импликација је истина.

Въпреки че горните четири изявления са изказани на различни езици, може да забележим (ако знаем съответните езици), че мисълта, която се съдържа в тях, е една и съща. Мисълта, която се съдържа в едно изявление, се нарича *съждение*; от тук идва и името на логиката — съждителна логика. Когато пък за едно съждение заявим, че е вярно, това наше заявление се нарича *твърдение*.

На разликите между изречение, изявление, съждение и твърдение се набляга във философската логика.* В математически контекст обаче тези разлики не са важни. Много често когато математици използват думата „твърдение“, те всъщност имат предвид това, което философите наричат „съждение“.

*Съответните английски термини са sentence, statement, proposition, assertion.

	отрицание
	„не“
p	$\neg p$
	$\sim p$
0	1
1	0

Таблица 1. Едноместна съждителна операция

		конюнкция	дизюнкция	импликация	еквиваленция
		„и“	„или“	„ако . . . , то . . . “	„тогава и само тогава, когато“
p	r	$p \,\&\, r$	$p \vee r$	$p \Rightarrow r$	$p \Leftrightarrow r$
		$p \wedge r$		$p \rightarrow r$	$p \leftrightarrow r$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Таблица 2. Двуместни съждителни операции

Булеви функции

Ако пренебрегнем философските въпроси, свързани със *съждителната логика*, то може да считаме, че тя представлява формален език за изразите, които използваме, когато дефинираме булеви функции. Да припомним, че *булева функция* означава, функция, чиито аргументи са елементи на множеството $\{0, 1\}$ и стойността ѝ също е елемент на $\{0, 1\}$.

В математическата логика, и в частност в съждителната логика, се използват малко по-различни означения за съждителните операции в сравнение с означенията, използвани в дискретната математика. Например отрицанието най-често се отбелязва посредством символите $\neg$ или $\sim$, а не с хоризонтална черта над израза (вж. таблица 1). Използваните означения за двуместните съждителни операции пък са дадени в таблица 2. Забележете, че конюнкцията се отбелязва посредством & или $\wedge$, а не като умножение.

В математическата логика съждителните операции имат следните приоритети:

– с най-висок приоритет е отрицанието $\neg$;

– със следващ по сила приоритет са конюнкцията & и дизюнкция-

та $\vee$;

– с най-слаб приоритет са импликацията $\Rightarrow$ и еквиваленцията $\Leftrightarrow$.

Например $x \Rightarrow \neg y_1 \vee y_2$ е същото като $x \Rightarrow ((\neg y_1) \vee y_2)$. Да забележим, че операциите $\&$ и $\vee$ са с един и същ приоритет, така че винаги трябва да използваме скоби, за да уточним коя от тях се извършва първо.* Аналогично е положението и при операциите $\Rightarrow$ и $\Leftrightarrow$.

Ето някои примерни булеви функции, дефинирани с помощта на означенията, използвани в логиката:

$$\begin{aligned} f_1(x, y) &= x \,\&\, \neg y \\ f_2(x, y) &= (x \,\&\, \neg y) \vee (\neg x \,\&\, y) \\ f_3(x, y, z) &= \neg(x \vee y \vee z) \end{aligned}$$

Да забележим, че съответствието между булеви функции и изрази не е взаимно еднозначно. Така например изразът $x \vee y$ може да се използва за дефинирането на следните различни по между си булеви функции:

$$\begin{aligned} f_1(x, y) &= x \vee y \\ f_2(y, x) &= x \vee y \\ f_3(x, y, z) &= x \vee y \end{aligned}$$

Както се вижда от дефиницията на функцията f_3, възможността да се използват фиктивни аргументи означава, че един израз може да се използва за дефиниране на безброй много булеви функции. Така например следващият списък съдържа само някои от безброй многото функции, които можем да дефираме с израза $x \vee y$:

$$\begin{aligned} h_1(x, y, z_1) &= x \vee y \\ h_2(x, y, z_1, z_2) &= x \vee y \\ h_3(x, y, z_1, z_2, z_3) &= x \vee y \\ h_4(x, y, z_1, z_2, z_3, z_4) &= x \vee y \\ h_5(x, y, z_1, z_2, z_3, z_4, z_5) &= x \vee y \\ &\dots \end{aligned}$$

*Това не е така в дискретната математика, където конюнкцията обикновено се обозначава като умножение и е с по-голям приоритет от дизюнкцията. Например в дискретната математика $x \vee y_1 y_2$ е същото като $x \vee (y_1 y_2)$, докато в логиката изразът $x \vee y_1 \,\&\, y_2$ е двусмислен.

Синтаксис на съждителната логика

По традиция, правилно построените изрази от някоя логика, вкл. от съждителната логика, се наричат *формули*. Това означава, че в математическата логика и логическото програмиране думата „формула“ има по-различен смисъл, отколкото в другите дялове на математиката.

Съждителната логика представлява формален език за изразите, които използваме, когато дефинираме булеви функции. Следователно формулите от съждителната логика представляват точно дефинирани низове от символи.* Добре е да обърнем по-специално внимание на това свойство на формулите. Да попитаме от колко символа е съставен неформалният израз $(x \,\&\, \neg y) \vee (\neg x \,\&\, y)$ е безсмислено, защото това е неформален израз, а не низ от символи. Формулите обаче, както и изобщо правилно построените изрази на кой да е формален език, са редици от символи и затова за тях не е безсмислено да се запитаме от колко символа са образувани.

Неформалните изрази, които използваме, когато дефинираме булеви функции, съдържат променливи като x, y, z и т. н. Следователно аналози на тези променливи ще трябва да има и във формалните изрази на съждителната логика. Ще ги наречем съждителни променливи.

2.2.1. Дефиниция. Нека **съжд** е сорт. Променливите от сорт **съжд** ще наречем *съждителни променливи*.

Нека $\neg$ е операционен символ от тип $\langle \mathbf{съжд} \rangle \mapsto \mathbf{съжд}$. Нека освен това имаме и три операционни символа $\&$, $\vee$ и $\Rightarrow$ от тип $\langle \mathbf{съжд}, \mathbf{съжд} \rangle \mapsto \mathbf{съжд}$. Използвайки тези символи, може да дефинираме съждителните формули по следния начин:

2.2.2. Дефиниция. Съждителните формули се дефинират индуктивно посредством следните три правила:

а) Ако $\mathtt{x}$ е съждителна променлива, то $\mathtt{x}$ е съждителна формула;

б) ако φ е съждителна формула, то низът $\neg\varphi$ е съждителна формула;

в) ако φ и ψ са съждителни формули, то низовете $(\varphi \,\&\, \psi)$, $(\varphi \vee \psi)$ и $(\varphi \Rightarrow \psi)$ са съждителни формули.

*Математиците обикновено говорят за думи, съставени от букви, вместо за низове от символи. Гьоте е казал, че математиците са като французите — ти им кажеш едно нещо, а те си го превеждат на собствения език и в крайна сметка се получава нещо съвсем различно. Например когато един дискретен математик говори за „букви“, той всъщност си мисли за произволни символи (напр. скоби или символи за логически операции). Когато същият математик говори за „думи“, той всъщност си мисли за низове и затова не се плаши от филологически абсурди като „празната дума“.

Да отбележим, че в така дадената дефиниция не се казва, че ако φ и ψ са формули, то низът $(\varphi \Leftrightarrow \psi)$ е съждителна формула. Направихме това умишлено, за да илюстрираме нещо, което логиците често правят — в дефиницията на формула те споменават само част от логическите операции, а за останалите приемат, че са съкращение. Например може да се уговорим, че когато пишем израз от вида $(\varphi \Leftrightarrow \psi)$, ние всъщност имаме предвид $((\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \varphi))$. По този начин, без да намаляваме изразителната сила на съждителната логика, ние можем да опростим формулировките на някои дефиниции, както и много от доказателствата на твърденията. Всъщност можеше още повече да опростим тази дефиниция като споменем в нея само две операции, например само операциите отрицание и конюнкция, защото всяка булева функция може да се изрази, използвайки единствено отрицание и конюнкция.*

***Задача* 4:** Нека $\mathtt{x}$ е съждителна променлива. Колко символа съдържа формулата $(\mathtt{x} \Leftrightarrow \mathtt{x})$?

В конкретния синтаксис на формулите може да се уговорим да пропускаме ненужните скоби, използвайки обичайния приоритет на съждителните операции.

***Задача* 5:** Нека $\mathtt{x}$, $\mathtt{y}$ и $\mathtt{z}$ са съждителни променливи. Намерете точния вид на следните формули, както и синтактичните им дървета: $\mathtt{x} \Rightarrow \mathtt{z}\,\&\,\mathtt{y}$, $\mathtt{x} \vee \mathtt{z} \Rightarrow \mathtt{y}$, $\neg(\neg\mathtt{x} \Rightarrow \neg(\mathtt{y}\ \&\ (\mathtt{z} \vee \mathtt{x}))\ \&\ \neg\mathtt{z})$.

Дефиницията на съждителна формула е дадена по такъв начин, че да направи вярно следното твърдение:

2.2.3. Твърдение за еднозначен синтактичен разбор. *Всяка съждителна формула притежава единствено синтактично дърво.*

От еднозначността на синтактичния разбор следва например, че никоя съждителна формула не може да бъде едновременно от вида $(\varphi_1\ \&\ \varphi_2)$ и $(\psi_1 \vee \psi_2)$, нито пък едновременно от вида $(\varphi_1 \Rightarrow \varphi_2)$ и $\neg\psi$. Също така от тук следва например, че една съждителна формула може да бъде едновременно от вида $(\varphi_1 \vee \varphi_2)$ и $(\psi_1 \vee \psi_2)$ само когато $\varphi_1 = \psi_1$ и $\varphi_2 = \psi_2$.

В дефиницията на съждителна формула използвахме обичайния инфиксен запис за двуместните операции и префиксен запис за отрицанието (вж. пример 2.1.7). Разбира се, не би било проблем да дадем

*По-нестандартно решение би било да използваме само стрелката на Пирс или само чертата на Шефер. В този случай можеше да минем с една единствена съждителна операция.

дефиницията и използвайки някой от другите записи. Това би било необичайно, но би свършило същата работа. Например

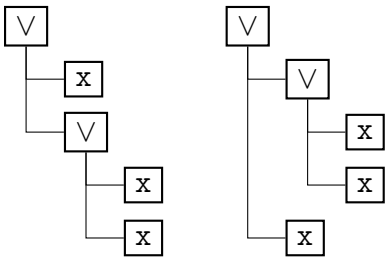
- за да получим съждителни формули в полски запис* (вж. пример 2.1.8), трябва да кажем че ако φ е формула, то $\neg\varphi$ е формула и ако φ и ψ са формули, то $\&\varphi\psi$, $\vee\varphi\psi$ и $\Rightarrow\varphi\psi$ са формули;
- за да получим съждителни формули в обратен полски запис (вж. пример 2.1.9), трябва да кажем че ако φ е формула, то $\varphi\neg$ е формула и ако φ и ψ са формули, то $\varphi\psi\&$, $\varphi\psi\vee$ и $\varphi\psi\Rightarrow$ са формули;
- за да получим съждителни формули в традиционен функционален запис (вж. пример 2.1.10), трябва да кажем че ако φ е формула, то $\neg(\varphi)$ е формула и ако φ и ψ са формули, то $\&(\varphi,\psi)$, $\vee(\varphi,\psi)$ и $\Rightarrow(\varphi,\psi)$ са формули.
- за да получим съждителни формули в модерен функционален запис (вж. пример 2.1.11), трябва да кажем че ако φ е формула, то $(\neg\varphi)$ е формула и ако φ и ψ са формули, то $(\&\varphi\psi)$, $(\vee\varphi\psi)$ и $(\Rightarrow\varphi\psi)$ са формули.

Обаче без значение кой точно запис решим да използваме, той задължително трябва да бъде такъв, че да осигурява еднозначен прочит на съждителните формули. Не е трудно с малки промени в дефиниция 2.2.2 да направим така, че твърдението за еднозначен синтактичен разбор да не е вярно. Нека например дефиницията казва, че ако φ и ψ са съждителни формули, то $\varphi\vee\psi$ също е съждителна формула. В такъв случай ако $\mathtt{x}$ е съждителна променлива, то $\mathtt{x}$ ще бъде съждителна формула, значи $\mathtt{x}\vee\mathtt{x}$ също ще бъде съждителна формула, а от тук може да получим, че и $\mathtt{x}\vee\mathtt{x}\vee\mathtt{x}$ е съждителна формула. Последният израз обаче няма еднозначен прочит, защото притежава две различни синтактични дървета, илюстрирани във фигура 10.

Много често логиците формулират твърдението за еднозначен синтактичен разбор, но го оставят без доказателство. За това има две причини.

Първата причина е следната. Когато изследваме някоя логика, и в частност съждителната логика, ние се интересуваме единствено от онези свойства, които по никакъв начин не зависят от това кой точно запис ще използваме (инфиксен, полски, обратен полски, функционален, някакъв друг). В частност всички формулировки на дефиниции и твърдения, както и разсъжденията, използвани в доказателствата, са

*Ян Лукашевич е използвал други означения за съждителните операции в своята версия на полския запис.



Фиг. 10. Две синтактични дървета за израза $\mathtt{x} \vee \mathtt{x} \vee \mathtt{x}$

такива, че те не се променят съществено, ако решим да сменим използвания запис на съждителните формули. Твърдението за еднозначност на синтактичния разбор обаче не може да се докаже без да се интересуваме от това какъв точно е използваният запис. Това означава, че когато доказва това твърдение, един логик трябва да се интересува от неща, за които обикновено не му се налага да се интересува.* И тъй като логикът не иска да се занимава с неща, с които обикновено не се занимава, той предпочита да си спести доказателството.

Втората причина е следната. Когато един логик остави еднозначността на синтактичния разбор недоказана, той все едно ни казва следното: „Дори да се окаже, че съждителните формули нямат еднозначен синтактичен разбор, защото съм сбъркал дефиницията на съждителна формула, нека считаме че вместо дадената тук дефиниция използваме някоя друга, при която има еднозначност на синтактичния разбор.“ Това не е некоректно поради следната причина: вече отбелязахме, че в математическата логика дефинициите, твърденията и доказателствата се правят по такъв начин, че те да не зависят от това какъв точно запис на формулите използваме. Следователно дори да сменим записа на формулите, няма да се наложи да поправяме нито едно доказателство.**

Ако дефинираме формулите да бъдат не редици от символи, а дървета, тогава отпада нуждата да доказваме, че всяка формула притежава единствено синтактично дърво, защото всяка формула съвпада със своето синтактично дърво.

*Например трябва да се интересуваме от поднизовете на една формула, от това кои от тези поднизове завършват с разделител, кои от префиксите на тези поднизове съдържат повече леви скоби, отколкото десни, и други подобни неща.

**Да отбележим, че дадената тук дефиниция 2.2.2 не е сбъркана и не се налага да я сменяме.

Семантика на съждителната логика

След като дефинирахме синтаксиса на съждителната логика, т.е. какво значи един израз да бъде правилно построена съждителна формула, трябва да дефинираме и семантиката на тази логика, т.е. какъв е смисълът на една съждителна формула. Смисълът на съждителната формула φ ще означаваме с $[\![\varphi]\!]$. Без ясно дефинирана семантика, съждителните формули биха били просто низове от символи и нищо повече.

Нека $\mathtt{x}$ и $\mathtt{y}$ са съждителни променливи. В такъв случай изразът $(\mathtt{x} \vee \mathtt{y})$ ще бъде съждителна формула. Да се запитаме какъв е смисълът на този израз.

Не можем просто да кажем, че $[\![\mathtt{x} \vee \mathtt{y}]\!]$ е функцията дизюнкция, защото трябва да уточним какви са аргументите на тази дизюнкция. Вече видяхме, че има безброй много булеви функции, които можем да дефинираме с израза $(\mathtt{x} \vee \mathtt{y})$ и всяка от тях е в някакъв смисъл дизюнкция. Би могло по някакъв, може би изкуствен начин да изберем една от всички тези безброй много булеви функции и да считаме, че $[\![\mathtt{x} \vee \mathtt{y}]\!]$ е така избраната булева функция. Правилното решение обаче се оказва друго: ще считаме, че смисълът $[\![\varphi]\!]$ на коя да е съждителна формула φ е функция, която зависи от стойностите на всички съждителни променливи, без значение дали се срещат във φ или не.

Тъй като $[\![\varphi]\!]$ трябва да зависи от стойностите на всички съждителни променливи, ще дефинираме понятието съждителна интерпретация:

2.2.4. Дефиниция. Казваме, че $\mathbf{I}$ е съждителна интерпретация, ако $\mathbf{I}$ е функция, чиято дефиниционна област е множеството на всички съждителни променливи, и $\mathbf{I}(\mathtt{x}) \in \{0, 1\}$ за всяка съждителна променлива $\mathtt{x}$.

Вече сме готови да дефинираме семантиката на съждителната логика:

2.2.5. Дефиниция. Нека $\mathbf{I}$ е коя да е съждителна интерпретация. *Стойността на съждителната формула* φ при интерпретация $\mathbf{I}$ ще означаваме с $[\![\varphi]\!]^{\mathbf{I}}$ и ще дефинираме по следния начин:

– ако $\mathtt{x}$ е съждителна променлива, то нека

$$[\![\mathtt{x}]\!]^{\mathbf{I}} = \mathbf{I}(\mathtt{x})$$

– за всяка съждителна формула φ, нека

$$[\![\neg\varphi]\!]^{\mathbf{I}} = \neg([\![\varphi]\!]^{\mathbf{I}}) \tag{1}$$

– за всеки две съждителни формули φ и ψ, нека

$$[\![\varphi \,\&\, \psi]\!]^{\mathbf{I}} = [\![\varphi]\!]^{\mathbf{I}} \,\&\, [\![\psi]\!]^{\mathbf{I}} \tag{2}$$

$$[\![\varphi \vee \psi]\!]^{\mathbf{I}} = [\![\varphi]\!]^{\mathbf{I}} \vee [\![\psi]\!]^{\mathbf{I}} \tag{3}$$

$$[\![\varphi \Rightarrow \psi]\!]^{\mathbf{I}} = [\![\varphi]\!]^{\mathbf{I}} \Rightarrow [\![\psi]\!]^{\mathbf{I}} \tag{4}$$

Обърнете внимание, че смисълът на знаците $\neg$, $\&$, $\vee$ и $\Rightarrow$ отляво и отдясно на равенствата (1)–(4) е много различен. Отляво на равенствата тези знаци са част от изрази, които са съждителни формули. Следователно тук знаците $\neg$, $\&$, $\vee$ и $\Rightarrow$ са просто символи — символи, които могат да бъдат въвеждани от клавиатурата, отпечатвани на компютърния екран и т. н. Отдясно на равенствата обаче тези знаци имат напълно различен смисъл. Тук те не са символи, а математически означения за булевите функции от таблици 1 и 2 на стр. 58.

Забележка: В езиците за програмирне разликата между символи и несимволи е много по-ясна. Например `1` е числото 1, а `"1"` е символът 1. Математиците обаче не са свикнали да ограждат с кавички символите и затова трябва да сме нащрек. Все пак нека запомним, че винаги когато използваме семантичните скоби $[\![\ ]\!]$, може да считаме, че изразът в тях е заграден в кавички, т. е. е редица от символи. Макар математиката да няма кавички, тя си има „антикавички“, защото функцията на скобите $[\![\ ]\!]$ е точно тази — да премахне кавичките: $[\![\texttt{"1"}]\!] = 1$.

2.2.6. Пример. Нека $\mathtt{x}$ и $\mathtt{y}$ са съждителни променливи, а съждителната интерпретация $\mathbf{I}$ е такава, че $\mathbf{I}(\mathtt{x}) = 1$ и $\mathbf{I}(\mathtt{y}) = 0$. Тогава:

$$[\![\mathtt{x}]\!]^{\mathbf{I}} = 1$$
$$[\![\neg\mathtt{x}]\!]^{\mathbf{I}} = 0$$
$$[\![\mathtt{y} \vee \mathtt{x}]\!]^{\mathbf{I}} = 1$$
$$[\![\neg\mathtt{y} \vee \mathtt{x}]\!]^{\mathbf{I}} = 1$$
$$[\![\neg(\mathtt{y} \vee \mathtt{x})]\!]^{\mathbf{I}} = 0$$

2.2.7. Дефиниция. а) Когато $[\![\varphi]\!]^{\mathbf{I}} = 1$, казваме, че съждителната формула е *вярна* при интерпретацията $\mathbf{I}$. Ако пък $[\![\varphi]\!]^{\mathbf{I}} = 0$, то казваме, че формулата е *невярна* при интерпретацията $\mathbf{I}$.

б) Някои съждителни формули са верни при всички интерпретации. Такива съждителни формули се наричат *съждителни тавтологии*.*

*Много често се срещат неправилни изписвания на думата тавтология, например „тафтология“ и „тавталогия“.

Пример. Нека $\mathtt{x}$ и $\mathtt{y}$ са съждителни променливи. Ето някои примерни съждителни тавтологии:

$$\mathtt{x} \,\&\, \mathtt{y} \Leftrightarrow \mathtt{y} \,\&\, \mathtt{x}$$
$$\mathtt{x} \vee \mathtt{x} \Leftrightarrow \mathtt{x}$$
$$\neg(\mathtt{x} \vee \mathtt{y}) \Leftrightarrow \neg\mathtt{x} \,\&\, \neg\mathtt{y}$$

Изводимост

Освен разнообразни логически езици, логиката изучава и принципите за правене на правилни доказателства. По отношение на съждителната логика това означава, че трябва да изследваме в кои случаи ако вече сме доказали, че формулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ са верни при някаква интерпретация, от тук ще можем да получим като следствие някоя нова формула ψ.

Изявление от вида

> ако формулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ са верни при някоя съждителна интерпретация, то формулата ψ също е вярна при тази интерпретация

ще наричаме *правило за извод* за съждителната логика.

Не е трудно да формулираме примерно правило за извод. Например ако вече сме доказали, че формулите φ и ψ са верни при някоя интерпратация, ясно е, че от тук ще следва, че формулата $\varphi \,\&\, \psi$ също е вярна. Това правило извод можем да запишем по следния начин:

$$\frac{\varphi \quad \psi}{\varphi \,\&\, \psi}$$

Изобщо използваме запис от вида

$$\frac{\varphi_1 \quad \varphi_2 \quad \ldots \quad \varphi_n}{\psi}$$

за да означим правилото, което казва, че ако вече сме доказали формулите над чертата, то ще можем да получим като следствие и формулата под чертата. Формулите над чертата се наричат *предпоставки* на правилото, а формулата под чертата — негово *заключение*.*

Древногръцкият логик Хризип формулирал следните пет правила за извод, които той наричал „примитивни“ правила.

*Вместо „предпоставка“ се използва и терминът „антецедент“, а вместо „заключение“ — „консеквент“ или „сукцедент“.

– Първото примитивно правило на Хризип е:

$$\frac{\varphi \Rightarrow \psi \quad \varphi}{\psi}$$

В съвременната логика това правило се нарича *modus ponens*.

– Второто примитивно правило на Хризип е:

$$\frac{\varphi \Rightarrow \psi \quad \neg\psi}{\neg\varphi}$$

В съвременната логика това правило се нарича *modus tollens*.

– Третото примитивно правило на Хризип е:

$$\frac{\neg(\varphi \,\&\, \psi) \quad \varphi}{\neg\psi}$$

– Четвъртото примитивно правило на Хризип е:

$$\frac{\neg(\varphi \Leftrightarrow \psi) \quad \neg\varphi}{\psi}$$

– Петото примитивно правило на Хризип е:

$$\frac{\varphi \vee \psi \quad \neg\varphi}{\psi}$$

***Забележка*:** Оригиналното четвърто правило на Хризип било формулирано с помощта на т. н. „изключващо или“. В петото правило също се говорело за изключващото, а не за обикновеното „или“. Обикновената дизюнкция, която използваме в съвременната математика, е била измислена след Хризип.

***Задача* 6:** По кое от правилата на Хризип се извършва всяко едно от следните разсъждения:

а) Ако вали дъжд, то времето е облачно. Но вали дъжд, значи времето е облачно.

б) Ако вали дъжд, то времето е облачно. Но времето не е облачно, значи не вали дъжд.

в) Не може едновременно да вали сняг и да има гръмотевици. Но вали сняг, значи няма гръмотевици.

г) В мача на шахматните претенденти (2006 г.) Камски печели тогава и само тогава, когато Топалов не печели. Но Камски не печели, значи Топалов печели.

д) Ще тичам в парка или ще ходя на планина. Но няма да тичам в парка, значи ще ходя на планина.

Освен петте „примитивни“ правила, Хризип е формулирал и четири начина, наречени от него „теми“, посредством които от петте правила можем да генерираме нови правила за извод. За да установи верността на някое разсъждение, Хризип постъпвал по следния начин: започвал да прилага четирите теми към разсъждението като по този начин го разбивал на все по-примитивни методи за разсъждение. Ако в крайна сметка всичко може да се сведе до петте примитивни правила за извод, значи разсъждението е вярно. Точният вид на четирите теми на Хризип не е известен със сигурност, но може да се реконструира. [6]

В съвременната логика не е обичайно от някакви примитивни правила за извод да получаваме по-сложни правила за извод, както е правел Хризип. Вместо това се използва точно определен набор правила за извод, които са такива, че да бъдат достатъчни сами по себе си, без да се налага генерираме нови правила за извод. Изследвани са различни видове системи за извод — от Хилбертов тип, Генценова секвенциална изводимост, естествена изводимост, резолютивна изводимост и др.

Ще опишем системите за извод от Хилбертов тип, които имат сравнително най-проста форма. Да наречем правило за извод, което има нула предпоставки, *аксиома*. Такова правило просто казва, че заключението му е винаги вярно. При системите за извод от Хилбертов тип се стремим да имаме само едно правило за извод, което не е аксиома — правилото *modus ponens*:

$$\frac{\varphi \Rightarrow \psi \quad \varphi}{\psi}$$

Следва примерен списък от аксиоми, които заедно с modus ponens са достатъчни, за да получим всички съждителни тавтологии.

– Аксиоми за импликацията:

$$(\varphi \Rightarrow (\psi \Rightarrow \chi)) \Rightarrow ((\varphi \Rightarrow \psi) \Rightarrow (\varphi \Rightarrow \chi))$$
$$\varphi \Rightarrow (\psi \Rightarrow \varphi)$$
$$((\varphi \Rightarrow \psi) \Rightarrow \varphi) \Rightarrow \varphi$$

– Аксиоми за конюнкцията:

$$\varphi \,\&\, \psi \Rightarrow \varphi$$
$$\varphi \,\&\, \psi \Rightarrow \psi$$
$$\varphi \Rightarrow (\psi \Rightarrow \varphi \,\&\, \psi)$$

– Аксиоми за дизюнкцията:

$$(\varphi \Rightarrow \chi) \Rightarrow ((\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi))$$
$$\varphi \Rightarrow \varphi \vee \psi$$
$$\psi \Rightarrow \varphi \vee \psi$$

– Аксиоми за отрицанието:

$$\neg\varphi \Rightarrow (\varphi \Rightarrow \psi)$$
$$(\varphi \Rightarrow \neg\varphi) \Rightarrow \neg\varphi$$

2.3. Логически запис на прости изявления на естествен език

Както отбелязахме в предния раздел, съждителната логика представлява формален език за изразите, които използваме при дефиниране на булеви функции. И въпреки че някои прости разсъждения могат да се извършат в рамките на съждителната логика, като цяло изразителната сила на тази логика е доста слаба. Не е лесно да се открие дори една единствена математическа теорема, която може да се докаже използвайки единствено средствата от съждителната логика. Да видим защо това е така.

В дефиниция 2.2.2 видяхме, че съждителните формули се формират от по-прости съждителни формули, използвайки съждителните операции $\neg$, $\&$, $\vee$ и $\Rightarrow$. А най-простите съждителни формули — тези, които не съдържат нито една съждителна операция — са съждителните променливи. Всяка съждителна формула може да бъде разпадната на съставните си части, като най-малките формули, от които е образувана тя, са съждителните променливи, които се съдържат в тази формула.

По-подобен начин може да анализираме не само формулите, но и изобщо всяко едно съждение. Във формулировката на всяко съждение може да открием по-прости съждения, които са свързани с логически операции като отрицание, конюнкция, дизюнкция, импликация и др. Само че докато най-простите съждителни формули — съждителните променливи — имат доста тривиална структура (по-точно нямат

никаква вътрешна структура), то дори да разпаднем формулировката на едно съждение на съставните му части и стигнем до по-прости съждения, които са изказани без да се използва нито една логическа операция, формулировките на тези по-прости съждения ще имат нетривиална вътрешна структура.

Да илюстрираме това с няколко примера на прости съждения, които са изказани без да се използва нито една логическа операция:

1. *Солон царува.*
2. *Сократ е смъртен.*
3. 79 *е четно число.*
4. $9 < 2$
5. *Орфей обича Евридика.*
6. *Елена се страхуваше от Менелай.*
7. *Аристотел преподава на Александър Македонски логика.*

От тези примери се вижда, че формулировката на всяко от тези съждения притежава вътрешна структура. В съждителната логика тази вътрешна структура се тривиализира — на всяко от горните съждения в съждителната логика би съответствала някоя съждителна променлива. Това е и основната причина за неадекватността на съждителната логика като език за записване на математически (и други) съждения.

Ако искаме някой логически език, да може да се използва за записване на твърдения и доказателства, то на този език ще трябва да можем да изразим вътрешната структура на съждения като горните седем. Нека я анализираме.

Първо, да забележим, че всяко едно от тези седем съждения е изказано посредством просто изречение. В традиционните теории за синтаксиса се счита, че простите изречения могат да съдържат части като сказуемо, подлог, допълнение, обстоятелство и др. От своя страна, всяка от тези части на изречението може да има свои собствени подчинени думи — определения. Например в изречението „Солон царува“ думата „царува“ е сказуемо, а „Солон“ — подлог. А в изречението „Аристотел преподава на Александър Македонски логика“ думата „преподава“ е сказуемото, „Аристотел“ — подлог, „логика“ — пряко допълнение и „Александър Македонски“ — непряко допълнение.

В по-модерните теории за синтаксиса се приема, че простото изречение може да съдържа следните части — предикат, аргументи и адюнкти. Сказуемото заедно със сказуемното определение образува предиката. Адюнкти са онези думи, за които има начин те да се премахнат от изречението и да се отделят в ново изречение, което не използва

същото сказуемо, без при това да се загуби изказаната информация. Останалите думи (всяка от които може да има собствени определения) са аргументите на предиката.

Да разгледаме следното примерно изречение:

Вчера Петкан разучаваше в общежитието езика пролог.

Тук сказуемото „разучаваше“ е предикатът. Думите „вчера“ и „в общежитието“ са адюнкти, защото могат да се отделят от основното изречение по следния начин:

Петкан разучаваше езика пролог.
Това той правеше вчера в общежитието.

Останалите части на изречението — „Петкан“ и „езика пролог“ — са аргументите на предиката „разучаваше“. Тези думи не могат да се отделят по естествен начин от основното изречение. Например:

Вчера разучаваше в общежитието езика пролог.
Това правеше Петкан.

Или:

Вчера Петкан разучаваше в общежитието.
Това правеше за езика пролог.

Броят на аргументите, които един предикат може да има, граматиците наричат *валентност* на предиката, а математиците — *арност* или *местност* на предиката. В естествените езици този брой зависи единствено от вида на използвания глагол. Например:

- Някои безлични глаголи нямат аргументи. Пример: „съмва се“.
- Непреходните глаголи имат един аргумент — подлогът. Пример: „царува“. Кой царува? Солон.
- При преходните глаголи аргументи са подлогът и допълненията. Пример: „обича“. Кой обича? Орфей. Кого обича? Евридика.
- При някои глаголи аргументите се свързват с разнообразни, но зависещи от конкретния глагол предлози. Пример: „страхува се“. От кого се страхува? От Менелай.*

В естествените езици от какъв тип е даден аргумент се определя по сложен начин въз основа на:

*Едно от най-трудните неща, когато се учи чужд език, е да се научи с какви предлози се свързват аргументите на различните глаголи. Въпреки че глаголите в различните езици използват различни предлози, кои точно са тези предлози не можем да разберем, поглеждайки в речника. Нужно е да четем разнообразни текстове и пак няма как да сме сигурни, че сме срещали всички видове аргументи, които може да има даден глагол.

– позицията на аргумента в изречението;

– падежът, в който е аргументът;

– залогът на сказуемото;

– използвания предлог;

– много други.

Например в изречението

Иван ухапа кучето.

думата „Иван“ е хапещият подлог, а „кучето“ — ухапаното допълнение, като това се разбира от позицията на тези думи спрямо сказуемото „ухапа“. Ако искаме да кажем, че кучето хапе, а Иван е ухапан, най-просто е да разменим думите „Иван“ и „кучето“, но това не е единственият начин. Например в разговорния български език можем да използваме енклитичното местоимение „го“ по следния начин:

Иван го ухапа кучето.

В книжовния български език пък можем да сменим залога на глагола от деятелен на страдателен:

Иван бе ухапан от кучето.

Вижда се, че правилата за определяне на вида на даден аргумент могат да бъдат доста сложни. Освен това тези правила са много различни в различните езици. За да си спестим тези големи усложнения, както и за да имаме логически език, който не е повлиян от това какъв е майчиният ни език, в логиката се използва следният запис за съждения, съставени от предикат и аргументи:

$$\textit{предикат}(\textit{аргумент}_1, \textit{аргумент}_2, \ldots, \textit{аргумент}_n)$$

При този запис от какъв тип е даден аргумент зависи единствено от това на коя позиция е сложен той. Няма нужда от падежи, предлози, глаголни залози и др.

Използвайки този запис, ето как можем да изкажем горните седем съждения:

1. царува(Солон)
2. смъртен(Сократ)
3. четно_число(79)
4. $<(9, 2)$
5. обича(Орфей, Евридика)
6. страхуваше_се(Елена, Менелай)
7. преподава(Аристотел, логика, Александър_Македонски)

Когато такива формулировки се използват в рамките на точно дефиниран логически език, те се наричат *атомарни формули*. Една особеност на атомарните формули, която ги отличава от естествените езици, е това, че при тях не се допускат липсващи аргументи. Това не е проблем, защото можем да „запълним“ липсващия аргумент, използвайки т. н. квантор за съществуване $\exists$. Например изявлението

Аристотел преподава логика.

може да се преведе на логически език по следния начин:

$\exists$`x` преподава(Аристотел, логика, `x`)

2.3.1. ***Забележка*:** В езика за програмиране пролог няма експлицитни квантори. От дясната страна на клаузите обаче се подразбира квантор за съществуване $\exists$ за всички променливи, които не се срещат никъде другаде. Затова тук не е проблем да имитираме липсващи аргументи. Ако пък искаме да покажем по-явно, че не се интересуваме от някой аргумент на предикат, пролог ни позволява да използваме символа `_` по следния начин:*

```
преподава(аристотел, логика, _)
```

От лявата страна на клаузите обаче, а също и при фактите, пролог подразбира квантор за всеобщност $\forall$. Затова тук е невъзможно да се използват липсващи аргументи. Не е възможно да обясним на пролог какво означава някой да преподава нещо, без при това да обясним също какво означава някой да преподава нещо някому. Например ако използваме клауза от вида

```
преподава(X, логика, _) :- .........
```

това, което ще обясним на пролог с тази клауза, не е при какви условия е вярно, че `X` преподава логика, а при какви условия е вярно, че `X` преподава логика всекиму и на всичко.

В много случаи съществителните и прилагателните имена в изречението се държат не като аргументи на предикат, а като отделни предикати. Да разгледаме изявлението

Иван намери мимоза.

Тук очевидно думата „Иван“ обозначава конкретно лице, така че това съществително е аргумент на предиката „намери“. Думата „мимоза“ обаче не обозначава конкретна мимоза, която Иван е намерил, а някаква неопределена мимоза. Следователно преводът

*Думата „аристотел“ е записана с малка буква, защото на пролог думите, започващи с главна буква, са променливи.

намери(Иван, мимоза)

е неправилен. Вместо това трябва да използваме по-сложна формулировка:

$\exists \mathtt{x}$ (намери(Иван, x) & мимоза(x))

Аналогично се тълкува изявлението

Иван намери цъфнала мимоза.

При него всяка от думите „намери“, „цъфнала“ и „мимоза“ е отделен предикат:

$\exists \mathtt{x}$ (намери(Иван, x) & мимоза(x) & цъфнало(x))

От друга страна, при изявлението

Иван намери срамежлива мимоза.

думата „срамежлива“ не е отделен предикат, защото словосъчетанието „срамежлива мимоза“ обозначава специален вид мимози, а не мимоза, която се срамува:

$\exists \mathtt{x}$ (намери(Иван, x) & срамежлива_мимоза(x))

Ако членуваме словосъчетанието „срамежлива мимоза“ така:

Иван намери срамежливата мимоза.

тогава то ще обозначава някоя конкретна срамежлива мимоза. Затова в този случай това словосъчетание не е отделен предикат, а аргумент на предиката „намери“:

намери(Иван, срамежливата_мимоза)

Членуваните съществителни или прилагателни в множествено число се превеждат с квантор за всеобщност $\forall$. Например изявлението

Шерлок Холмс залови престъпниците.

се превежда така:

$\forall \mathtt{x}$ (престъпник(x) $\Rightarrow$ залови(Шерлок_Холмс, x))

т.е. все едно казваме „за всяко x, ако x е престъпник, то x бе заловен от Шерлок Холмс.“

Да обобщим казаното със следните правила:

– Нечленувани съществителни и прилагателни в единствено или множествено число* се превеждат с квантор за съществуване $\exists$. Пример: „Учен намери *срамежлива мимоза*“ — някой учен намерил някаква срамежлива мимоза.

*Въпреки множественото число, обикновено логическият смисъл на изявлението „*Иван намери цъфнали мимози*“ е същият, както ако изречението бе формулирано с единствено число.

– Съществителни собствени, както и членувани прилагателни и съществителни в единствено число са аргументи на предикат. Пример: „Иван намери срамежливата мимоза“.
– Членувани съществителни и прилагателни в множествено число се превеждат с квантор за всеобщност $\forall$. Пример: „Срамежливите мимози са мимози“, т.е. всички представители на вида „срамежлива мимоза“ спадат към рода „мимоза“ (на семейство Fabaceae, подсемейство Mimosaceae).

Във всички разгледани примери аргументите на предикатитите се оказваха или конкретни обекти (Сократ, Евридика, Шерлок_Холмс), или променливи ($\mathtt{x}, \mathtt{y}, \mathtt{z}$). Има случаи, когато това не е достатъчно и се налага да използваме по-сложни изрази, наречени *термове*.* Да разгледаме следното изявление:

Фредерик Кайо откри столицата на Нубия.

Тук словосъчетанието „столицата на Нубия“ е аргументът на предиката „откри“. Това словосъчетание логиците превеждат с помощта на следния терм:

столицата_на(Нубия)

Следователно тук използваме израза „столицата на“ сякаш е функция, която има за аргумент „Нубия“, и връща като стойност столицата на Нубия. Затова горното изявление може да се преведе по следния начин:

открИ(Фредерик_Кайо, столицата_на(Нубия))

Ето пример с по-сложен терм. Изявлението

Баща на баща е дядо.

може да се преведе така:

$\forall\mathtt{x}$ дядо(бащата_на(бащата_на($\mathtt{x}$)), $\mathtt{x}$)

2.4. Предикатна логика

Уводни бележки

В предния раздел видяхме, че за да може на един логически език да формулираме разнообразни съждения, в този език задължително трябва да има атомарни формули, например

просто_число(60)

*По-простите изрази като „Сократ“ и променливата „$\mathtt{x}$“ също се считат за вид термове.

Освен това желателно е в този език да има и сложни термове, например

бащата_на(бащата_на(Зевс))

За да бъде удобно използването на логическия език, той трябва да поддържа различни сортове. Например със символа „60“ може да означаваме индивид от сорт **int**, а със символа „Зевс“ — индивид от сорт **персонаж**. Ако направим това, ще получим това, което логиците наричат *многосортна предикатна логика*. За да опростим нещата обаче, в този курс ще дефинираме едносортната предикатна логика, в която има само два сорта — **индив** и **съжд**. Сортът **индив** се използва за всички индивиди, за които формулираме съжденията, без значение какви са те. Например числото 60 и бог Зевс са индивиди от сорт **индив**. А сортът **съжд** се използва за съжденията.

Използвайки сортовете **индив** и **съжд**, не е трудно да определим какви трябва да бъдат типовете на символите, които използвахме в предния раздел. Ето няколко примера:

1. В терма „бащата_на(`x`)“ символът `x` е променлива. Подразбираната стойност на тази променлива обаче не е съждение, както беше при съждителната логика, а индивид. Следователно тук `x` е променлива от сорт **индив**.
2. Символите „Солон“, „Сократ“, „79“, „9“, „2“, „Орфей“, „Евридика“, „Елена“, „Менелай“, „Аристотел“, „Александър_Македонски“ и „логика“ означават индивиди, които не зависят от никакви аргументи. Следователно техният тип е

   $$\langle\rangle \mapsto \textbf{индив}$$

   Такива символи логиците наричат символи за индивидни константи.
3. Символите „бащата_на“ и „столицата_на“ означават индивиди, които зависят от един аргумент. Следователно типът на тези символи е

   $$\langle \textbf{индив} \rangle \mapsto \textbf{индив}$$

   Такива символи логиците наричат функционални символи.
4. Във формулите „царува(Солон)“ и „преподава(Аристотел, Александър_Македонски, логика)“ символите „царува“ и „преподава“ означават съждения, които зависят от аргументи, съответно един и три. Следователно типовете на символите „царува“ и „преподава“ са съответно

   $$\langle \textbf{индив} \rangle \mapsto \textbf{съжд}$$

   $$\langle \textbf{индив}, \textbf{индив}, \textbf{индив} \rangle \mapsto \textbf{съжд}$$

Такива символи логиците наричат предикатни символи.

Да дадем и точна дефиниция за видовете символи, които разгледахме.

✓ **2.4.1. Дефиниция.** а) Да фиксираме от тук до края на този курс два сорта **индив** и **съжд**.

б) Променливите от сорт **индив** се наричат *индивидни променливи*.

в) Символите, чийто тип е

$$\langle\rangle \mapsto \textbf{индив}$$

се наричат *символи за индивидни константи*.

г) Символите, чийто тип е

$$\langle \underbrace{\textbf{индив}, \textbf{индив}, \ldots, \textbf{индив}}_{n \text{ броя } \textbf{индив}} \rangle \mapsto \textbf{индив}$$

се наричат *n-местни функционални символи*.

д) Символите, чийто тип е

$$\langle \underbrace{\textbf{индив}, \textbf{индив}, \ldots, \textbf{индив}}_{n \text{ броя } \textbf{индив}} \rangle \mapsto \textbf{съжд}$$

се наричат *n-местни предикатни символи*.

Забележка: Важно е да не забравяме, че символите за индивидни константи, функционалните символи и предикатните символи са символи, които сами по себе си нямат фиксиран смисъл. Също както π не е число, а буква от гръцката азбука, с която означаваме определено число, така и символът „Солон“ не е цар на Атѝна, а означение, което можем да използваме, за да назоваваме този цар. Обаче нищо не пречи да използваме означението „Солон“ и за напълно различен индивид, например за числото 732691.

Тъй като в предикатната логика няма съждителни променливи, то за краткост, вместо „индивидна променлива“, може да казваме просто *променлива*.

Също така, вместо термина „символ за индивидна константа“ може да използваме и по-кратките изрази *символ за константа*, *индивидна константа* и дори само *константа*. Това е възможно, тъй като в контекста на логическото програмиране и математическата логика други константи почти няма и значи винаги когато кажем, че нещо е „константа“, е ясно, че всъщност искаме да кажем, че то е „символ за индивидна

константа“. В това отношение функционалните и предикатните символи са „онеправдани“, защото в логическото програмиране се използват функции и предикати и затова не е правилно вместо „функционален символ“ да казваме накратко „функция“, нито пък вместо „предикатен символ“ може да казваме „предикат“.*

Някои математици считат, че символите за индивидни константи всъщност са специален вид функционални символи, а именно — нулместни функционални символи (т. е. с нула аргументи). При други математици пък функционалните символи задължително имат поне един аргумент и значи символите за индивидни константи не са функционални символи.

За константите (или нулместните функционални символи), както и за нулместните предикатни символи казваме, че са *нуларни*. За едноместните функционални и предикатни символи казваме, че са *унарни*. За двуместните функционални и предикатни символи казваме, че са *бинарни*. За триместните функционални и предикатни символи казваме, че са *тернарни*. Изобщо, за n-местните функционални и предикатни символи казваме, че са n-арни, а n е тяхната *арност*** или *местност*.

✓ **2.4.2. Уговорка.** За да не се налага винаги изрично да се уточнява кой символ е константа, кой функционален, кой предикатен и кой променлива, в математиката се използват следните уговорки:

- за променливи се използват буквите $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}$;
- буквите $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}$ са символи за индивидни константи;
- за функционални (не нулместни) символи се използват буквите $\mathtt{f}, \mathtt{g}, \mathtt{h}$, а при нужда още и $\mathtt{j}, \mathtt{k}, \mathtt{l}$.
- за предикатни символи се използват буквите $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$.

Ако трябват повече символи, се използват индекси, примове и т. н., напр. $\mathtt{x}$, $\mathtt{x}'$, $\mathtt{x}_3$ и $\mathtt{x}''_2$ са променливи, а $\mathtt{p}$, $\mathtt{p}_1$ и $\mathtt{p}''_2$ са предикатни символи.

Благодарение на тази уговорка, когато видим израз като $\mathtt{f}(\mathtt{g}(\mathtt{c}), \mathtt{x}, \mathtt{b})$, е ясно, че това е терм, а не атомарна формула (защо?). В този терм $\mathtt{f}$ е

*На изпита по логическо програмиране се счита за груба грешка, ако за нещо, което е функционален символ, се каже че е функция.

Този смисъл на думата „арност“ се различава от дадения в дефиниция 2.1.2 г), според който арността на например един двуместен функционален символ би била не числото 2, а двойката $\langle$индив**, **индив**$\rangle$. Тази двойна употреба не води до проблеми, защото в едносортната предикатна логика сортовете на аргументите винаги са **индив**. Затова ако например кажем, че арността е 2, от тук е ясно, че другият вид арност е $\langle$**индив**, **индив**$\rangle$. И обратно, ако кажем, че арността е $\langle$**индив**, **индив**$\rangle$, то от това става ясно, че числовата арност е 2.

триместен функционален символ, g е едноместен функционален символ, c и b са символи за константи и x е променлива. От друга страна изразът p(f(g(x), c)) е атомарна формула, а не терм. В нея p е едноместен предикатен символ, c е символ за константа, f е двуместен функционален символ, а g е едноместен функционален символ.

✓ ***Задача* 7:** Открийте функционалните символи, константите и променливите в терма f(f(c, a), f(g(g(x)), h(y))). За всеки от функционалните символи определете неговата арност.

✓ ***Задача* 8:** Кои от следните изрази не са термове и защо: f(c), c(f), c, f, x(c), f(f(x)), f(f(f(c))), f(f(f(c, c))), g(g(x, x), g(x, y)).

✓ ***Задача* 9:** Кои от следните изрази са атомарни формули и кои от тях не съдържат променливи: c, f(c), p(c), p(f(f(f(c)))), p(f(x)), f(p(x)), p(p(x)), f(f(x))?

Сигнатури

Използвайки символите, които дефинирахме в предния подраздел, можем да дадем точна дефиниция и на понятията терм и предикатна формула. За много приложения обаче е удобно да ограничим термовете и формулите да не използват произволни символи, а само символи, включени в определен от нас списък. Ще наречем този списък сигнатура.

2.4.3. Дефиниция. Крайна или безкрайна редица от символи за индивидни константи, функционални символи и предикатни символи се нарича *сигнатура*.*

В зависимост от това какви символи включим в сигнатурата, ще получим най-различни, нееквивалентни по между си варианти на предикатната логика. При някои сигнатури например може да няма функционални символи, а при други да има. При някои има само един предикатен символ, който е едноместен, при други само един, който е двуместен, при трети имаме безброй много триместни предикатни символи и т. н. Изразителната сила на предикатната логика е различна при всяка една от тези сигнатури. Ще използваме фрази като „терм τ при

*Терминът сигнатура (на англ. signature) се използва в математическата логика, алгебрата, теория на езиците за програмиране и информатиката. Понякога в математическата логика и алгебрата вместо за сигнатура се говори за *език* (language). В универсалната алгебра сигнатурите понякога се наричат *типове* (types). В теория на моделите пък се използва терминът *речник* (vocabulary).

сигнатура $\mathfrak{sig}$“ или „$\mathfrak{sig}$-терм τ“ и „атомарна формула φ при сигнатура $\mathfrak{sig}$“ или „$\mathfrak{sig}$-формула φ“, за да кажем, че термът τ и атомарната формула φ са построени, използвайки символите, предоставени от сигнатурата $\mathfrak{sig}$.

За да не усложняваме излишно формулировките на дефинициите и твърденията, нека фиксираме една конкретна сигнатура $\mathfrak{sig}$. Ако в някоя дефиниция или твърдение говорим просто за „функционални символи“, „променливи“, „термове“ и т. н., без да е споменато изобщо за сигнатура, то ще имаме предвид функционални символи, променливи и термове при така фиксираната сигнатура $\mathfrak{sig}$.

Термове

Вече сме готови да дадем и дефиницията на терм. Дефиницията ще бъде индуктивна. Това означава, че ако за даден израз успеем да докажем, че е терм, използвайки в доказателството единствено трите правила от дефиниция 2.4.4, то тогава този израз е терм. Ако пък за даден израз е невъзможно да се докаже, че е терм, използвайки единствено тези три правила, то тогава той не е терм.

✓ **2.4.4. Дефиниция.** Нека $\mathfrak{sig}$ е сигнатура. Понятието *терм* при сигнатура $\mathfrak{sig}$ или $\mathfrak{sig}$-терм се дефинира индуктивно посредством следните правила:

а) Ако $\mathtt{x}$ е индивидна променлива, то $\mathtt{x}$ е $\mathfrak{sig}$-терм.

б) Ако $\mathtt{c}$ е символ за константа от $\mathfrak{sig}$, то $\mathtt{c}$ е $\mathfrak{sig}$-терм.

в) Ако $\mathtt{f}$ е n-местен функционален символ от $\mathfrak{sig}$ и $\tau_1, \tau_2, \ldots, \tau_n$ са $\mathfrak{sig}$-термове, то низът $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ е $\mathfrak{sig}$-терм.

2.4.5. ***Забележка:*** Вижда се, че в така дадената дефиниция използваме традиционен функционален запис за термовете (вж. пример 2.1.10). Нямаше да има никакъв проблем в нея да използваме кой да е друг запис, който гарантира еднозначност на синтактичния разбор. Например ако искахме да ползваме термове в полски запис, в 2.4.4 в) вместо за $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ можеше да кажем за низа $\mathtt{f}\tau_1\tau_2\ldots\tau_n$, че е $\mathfrak{sig}$-терм.

✓ ***Задача* 10:** Нека $\mathtt{x}$ е променлива, а сигнатурата $\mathfrak{sig}$ включва символ за константа $\mathtt{c}$ и двуместен функционален символ $\mathtt{f}$. Използвайки правилата от дефиниция 2.4.4, докажете, че $\mathtt{f}(\mathtt{c}, \mathtt{f}(\mathtt{c}, \mathtt{x}))$ е $\mathfrak{sig}$-терм.

***Задача* 11:** Съществуват ли термове:

– които съдържат скоби, но не съдържат запетаи?

– които съдържат запетаи, но не съдържат скоби?

– в които броят на левите скоби е по-голям от броя на десните скоби?

***Задача* 12:** Напишете терм, който съдържа 2 скоби и 6 запетаи. Напишете терм, който съдържа 6 скоби и 2 запетаи.

****Задача* 13:** Да допуснем, че дефиницията позволяваше специалните символи да се използват като константи и функционални символи. Нека лявата скоба е символ за константа, дясната — едноместен функционален символ, а запетаята — двуместен функционален символ. Открийте функционалните символи и символите за константи в „термовете“, оградени с правоъгълници:

`(` `)(()` `)()(())` `,((,()` `,()((),,((,())`

Един любопитен факт с дълго доказателство е това, че подобни изрази винаги притежават еднозначен прочит — никога не възниква ситуация, при която някоя скоба е възможно да се изтълкува хем като истинска скоба, хем като функционален символ. Това отчасти се използва в езика пролог, в който символът запетая се използва хем като разделител на аргументите на функционалните символи, хем като двуместен функционален символ.

В раздела за формални езици (вж. раздел 2.1) споменахме, че формалните езици се делят на два вида — такива, за които е предназначено да се обработват от компютър, и такива, които са предназначени за хора. Също така споменахме, че когато формалният език е предназначен за хора, не е нужно изразите от езика да се дефинират по начин, който е удобен за използване от хора. Този парадокс обяснихме по следния начин: щом като изразите от формалния език ще се четат от хора, значи няма да възникнат проблеми, ако ги пишем не съвсем според дефиницията, но когато ги четем, се сещаме кой точно формален израз имаме предвид. При езиците, обработвани от компютър, не може да постъпваме по този начин, защото компютърът не може сам да поправя допуснатите от нас волни или неволни синтактични грешки.

В дадената току-що дефиниция 2.4.4 термовете не са дефинирани по начин, удобен за използване от хора. Това е така, защото за по-лесно ще дефинираме езика на предикатната логика като език за хора, а не като език за компютри. Ако искахме да дефинираме език, който ще се обработва от компютри, тогава например:

– символите за константи и функционалните символи всъщност не биха били единични символи, а низове от символи, защото при сегашната дефиниция „Солон“, „Орфей“ и т. н. не са термове;

– за по-добра четливост бихме позволявали използването на интервали в термове от вида $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$;

– в някои случаи бихме допускали инфиксен запис, напр. $\mathtt{a}+\mathtt{b}$ вместо $+(\mathtt{a}, \mathtt{b})$.

Но въпреки че нашата дефиниция не споменава изрично всички тези „удобства“, няма никакви причини да не се възползваме от тях. Например, спокойно може да пишем термове като $\mathtt{a} + \mathtt{b}$, стига да се сещаме, че това всъщност е термът $+(\mathtt{a}, \mathtt{b})$.

✓ ***Задача* 14:** Колко скоби се съдържат в терма $\mathtt{b} + \mathtt{x} * \mathtt{a}$?

***Задача* 15:** Нека сигнатурата $\mathfrak{sig}$ е такава, че в нея няма нито един символ за константа. Използвайки индуктивния принцип за термове, докажете, че всеки $\mathfrak{sig}$-терм съдържа поне една променлива.

***Задача* 16:** Нека сигнатурата $\mathfrak{sig}$ е такава, че всички функционални символи от $\mathfrak{sig}$ имат арност 2 (или 0). Използвайки индуктивния принцип за термове, докажете, че във всички термове при сигнатура $\mathfrak{sig}$ броят на запетаите е равен на броя на десните скоби.

Атомарни формули

Атомарните формули при сигнатура $\mathfrak{sig}$ може да дефинираме така:

✓ **2.4.6. Дефиниция.** Ако $\mathtt{p}$ е n-местен предикатен символ от $\mathfrak{sig}$ и $\tau_1, \tau_2, \ldots, \tau_n$ са $\mathfrak{sig}$-термове, то низът $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ е *атомарна формула* при сигнатура $\mathfrak{sig}$ (или атомарна $\mathfrak{sig}$-формула).

2.4.7. Означение. За удобство атомарните формули, образувани от нулместен предикатен символ, обикновено се записват без скоби, т.е. $\mathtt{p}$, а не $\mathtt{p}()$. Това е аналогично на термовете, образувани от нулместни функционални символи, т.е. символи за константи. Например ако $\mathtt{c}$ е символ за константа, то термът, образуван от $\mathtt{c}$ е просто $\mathtt{c}$, а не $\mathtt{c}()$.*

Просто сравнение на тази дефиниция с дефиницията на терм (вж. точка 2.4.4 в) от дефиницията) показва, че една атомарна формула $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ прилича по всичко на терм с единствената разлика, че

*Причината, поради която дефиниция 2.4.6 не е съобразена с този начин за запис, е това, че така си спестяваме необходимостта в различните доказателства да разглеждаме случаи според арността на предикатните символи. Впрочем тук не сме напълно последователни, защото ако в дефиницията за терм не бяхме отделили символите за константи като отделен случай, а считахме, че те са нулместни функционални символи, то бихме си опростили някои от доказателствата още повече.

$\mathtt{p}$ не е функционален, а предикатен символ. И също както при термовете, когато например + е двуместен функционален символ, се уговорихме за удобство да използваме инфиксен запис като пишем например $\mathtt{a}+\mathtt{b}$ вместо $+(\mathtt{a},\mathtt{b})$, така и при атомарните формули при някои предикатни символи за удобство ще използваме инфиксен запис. Например ако „$<$“ е двуместен предикатен символ, ще пишем $\mathtt{a}<\mathtt{b}$ вместо $<(\mathtt{a},\mathtt{b})$ и също ако „$=$“ е двуместен предикатен символ, ще пишем $\mathtt{a}=\mathtt{b}$ вместо $=(\mathtt{a},\mathtt{b})$. Разбира се, това правим само за наше удобство. Правилните атомарни формули, отговарящи на дефиниция 2.4.6, са $<(\mathtt{a},\mathtt{b})$ и $=(\mathtt{a},\mathtt{b})$.

Тъй като атомарните формули приличат на термове, може да възникне въпросът защо изобщо имаме нужда от това понятие. Не може ли да считаме символите $=$ и $<$ за функционални, при което $\mathtt{a}=\mathtt{b}$ и $\mathtt{a}<\mathtt{b}$ ще станат термове и няма да имаме нужда от атомарни формули?

Отговорът на този въпрос е следният: въпреки че като запис термовете и атомарните формули си приличат, сортовете ги различават. Термовете са синтактични обекти, чиято стойност е от сорт **индив**, а атомарните формули — синтактични обекти, чиято стойност е от сорт **съжд**. Например може да сложим терм като аргумент на функционален символ и тогава получаваме по-голям терм. Също може да сложим термове като аргумент на предикатен символ и в резултат получаваме атомарна формула. Обаче ако сложим атомарна формули като аргумент на функционален или предикатен символ, резултатът е безсмислица.

✓ **2.4.8. Пример.** Ето примерен терм, който се получава като дадем по-малкия терм „Зевс“ като аргумент на функционалния символ „баща_на“:

баща_на(Зевс)

Ето примерна атомарна формула, която се получава като дадем термовете „баща_на(Зевс)“ и „Прометей“ като аргументи на двуместния предикатен символ „чичо“:

чичо(баща_на(Зевс), Прометей)

Ето примерна безсмислица, която се получава като дадем атомарна формула като аргумент на функционален символ:

баща_на(чичо(Кронос, Прометей))

Интуитивно атомарната формула „чичо(Кронос, Прометей)“ казва, че Кронос е чичо на Прометей. Т.е. това е някакво съждение. Кой е бащата на това съждение? Разбира се, това е безсмислен въпрос. Съжденията могат да бъдат например верни и могат също да бъдат неверни, но

нямат бащи. Ето още една безсмислица, която се получава като дадем атомарна формула като аргумент на предикатен символ:

чичо(Хиперион, чичо(Кронос, Прометей))

Чий чичо е Хиперион? На съждението „Кронос е чичо на Прометей“.

✓ **2.4.9. Пример.** Да разгледаме следните три израза, в които 0 е символ за константа, „+“ е двуместен функционален символ и = е двуместен предикатен символ:

$$0 + (0 + 0)$$
$$0 = (0 + 0)$$
$$0 + (0 = 0)$$

Първият от тези изрази е терм. Например ако интерпретираме символа 0 като числото нула, а „+“ като събиране на числа, тогава стойността на този терм е нула. Вторият от тези изрази е атомарна формула. Ако отново интерпретираме символа 0 като числото нула, „+“ като събиране на числа и „=“ като равенство, тогава тази атомарна формула е вярна. Третият от горните изрази обаче е безсмислица. Не може да съберем $0 = 0$ с числото нула.*

Предикатни формули

В раздел 2.3 вече видяхме, че много съждения не могат да се изразят, използвайки само средствата, с които разполага съждителната логика. Например съждения като „Сократ е смъртен“ и „Орфей обича Евридика“ е най-естествено да се запишат като атомарни формули. Обаче не може да се ограничим само с атомарните формули, защото много съждения се формулират, използвайки различни логически операции, а в атомарните формули няма логически операции. Да разгледаме например следното съждение:

Ако Афродита е най-красивата, то Елена ще обича Парис.

Използвайки средствата на съждителната логика, можем да представим това съждение посредством формулата $\mathtt{x} \Rightarrow \mathtt{y}$, където $\mathtt{x}$ е съждителна променлива, с която означаваме съждението „Афродита е най-красивата“, а $\mathtt{y}$ съждителна променлива, с която означаваме съждението „Елена обича Парис“. Съждителните променливи $\mathtt{x}$ и $\mathtt{y}$ обаче по

*В някои езици за програмиране, напр. си, този израз не е безсмислен (по-точно изразът `0+(0==0)`). Това е така само защото в тези езици равенството не е логическа, а аритметична операция, чиято стойност е числото едно или нула, а не истина или лъжа.

никакъв начин не показват какви съждения сме означили с тях. От друга страна, използвайки атомарни формули, можем да преведем не цялото съждение, а само поотделно двете му части, например така:

най-красива(Афродита)
ще_обича(Елена, Парис)

Възниква въпросът — не може ли да дефинираме такава логика, в която хем имаме атомарни формули, хем можем да използваме и съждителни операции, както в съждителната логика? Отговорът на този въпрос е положителен.

Да си припомним дефиницията на съждителна формула (дефиниция 2.2.2). Тя бе индуктивна като най-простите съждителни формули са съждителните променливи, а от произволни вече построени съждителни формули можеше да образуваме нова съждителна формула, използвайки съждителните операции $\neg$, $\&$, $\vee$ и $\Rightarrow$. Например ако вече знаем, че φ и ψ са съждителни формули, то тогава може да заключим, че низът $(\varphi \,\&\, \psi)$ също е съждителна формула.

Ако в момента дадем аналогична дефиниция, само че вместо съждителни променливи използваме атомарни формули, ще получим това, което в логиката се нарича „безквънторна предикатна формула“. Използвайки безквънторна предикатна формула, горното съждение може да бъде преведено ето така:

най-красива(Афродита) $\Rightarrow$ обича(Елена, Парис)

Много неща могат да се направят, използвайки безквънторни предикатни формули. Сега обаче ще дадем направо по-общата дефиниция за произволна предикатна формула. Разликата между дефиницията на произволна предикатна формула и дефиницията на безквънторна предикатна формула се състои в това, че при произволните предикатни формули освен съждителните операции $\neg$, $\&$, $\vee$, $\Rightarrow$ и $\Leftrightarrow$ може да се използват и кванторите $\forall$ и $\exists$.

Нека $\bot$ е операционен символ от тип $\langle\rangle \mapsto$ **съжд**, който няма да считаме за нулместен предикатен символ.* Ще го използваме, за да означим формула, която е винаги невярна. Нека $\&$, $\vee$ и $\Rightarrow$ са операционни символи от тип $\langle$**съжд**, **съжд**$\rangle \mapsto$ **съжд**. Използвайки тези символи, може да дефинираме предикатните формули по следния начин:

*Всъщност това означава, че тук „модифицираме“ дефиниция 2.4.1 д). Ще считаме, че предикатни са всички операционни символи, които са различни от $\bot$ и чийто тип е

$$\langle \underbrace{\textbf{индив}, \textbf{индив}, \ldots, \textbf{индив}}_{n \text{ броя } \textbf{индив}} \rangle \mapsto \textbf{съжд}$$

✓ **2.4.10. Дефиниция.** *Предикатните формулите* при сигнатура $\mathfrak{sig}$ или просто $\mathfrak{sig}$-*формулите* се дефинират индуктивно:

а) ако φ е атомарна $\mathfrak{sig}$-формула, то φ е $\mathfrak{sig}$-формула;

б) $\bot$ е $\mathfrak{sig}$-формула;

в) ако φ и ψ са $\mathfrak{sig}$-формули, то низовете $(\varphi \,\&\, \psi)$, $(\varphi \vee \psi)$ и $(\varphi \Rightarrow \psi)$ са $\mathfrak{sig}$-формули;

г) ако $\mathtt{x}$ е променлива и φ е $\mathfrak{sig}$-формула, то низовете $\forall \mathtt{x}\, \varphi$ и $\exists \mathtt{x}\, \varphi$ са $\mathfrak{sig}$-формули.

Разбира се, при различните сигнатури имаме различни атомарни формули, а значи при различните сигнатури ще имаме и различни формули. Множеството от всички формули при някоя конкретна сигнатура наричаме *език на предикатната логика* или просто *език*.

2.4.11. Когато се дава точната дефиниция на понятието „формула“, не е нужно в нея да се споменават изрично всички логически операции. Например в дефиниция 2.4.10 не сме казали нищо за операцията еквиваленция. Това не създава проблеми, защото можем да считаме, че всяка формула от вида $\varphi \Leftrightarrow \psi$ представлява съкратен запис на формулата $((\varphi \Rightarrow \psi) \,\&\, (\psi \Rightarrow \varphi))$. Също така не сме споменали и отрицанието. Това също не създава проблеми, защото може да считаме, че всяка формула от вида $\neg\varphi$ представлява съкратен запис на формулата $(\varphi \Rightarrow \bot)$. Удобно е също така да считаме, че $\top$ е съкратен запис на формулата $\neg\bot$, т. е. на формула, която е винаги вярна.

✓ **2.4.12. Пример.** Нека $=$ и $\in$ са двуместни предикатни символи, а $\mathtt{x}$, $\mathtt{y}$ и $\mathtt{z}$ са променливи. Тогава изразът*

$$\forall \mathtt{x}\, \forall \mathtt{y}\, (\forall \mathtt{z}\, (\mathtt{z} \in \mathtt{x} \Leftrightarrow \mathtt{z} \in \mathtt{y}) \Rightarrow \mathtt{x} = \mathtt{y})$$

е съкратен запис на формулата

$$\forall \mathtt{x}\, \forall \mathtt{y}\, (\forall \mathtt{z}\, ((\in(\mathtt{z}, \mathtt{x}) \Rightarrow \in(\mathtt{z}, \mathtt{y})) \,\&\, (\in(\mathtt{z}, \mathtt{y}) \Rightarrow \in(\mathtt{z}, \mathtt{x}))) \Rightarrow =(\mathtt{x}, \mathtt{y}))$$

✓ **2.4.13.** За улеснение, когато записваме формули може да използваме по-неформален запис. Вече споменахме, че атомарните формули от вида $=(+(0, 0), 0)$ ще бъдат записвани по-четливо като $0 + 0 = 0$. Освен това ще си позволяваме да изпускаме някои от скобите. Например може да считаме, че всяка формула от вида $\varphi_1 \vee \varphi_2 \vee \varphi_3 \vee \varphi_4$ е съкратен запис

*В теория на множествата този израз се нарича аксиома за екстенсионалност или аксиома за обемност.

на формулата $(((\varphi_1 \vee \varphi_2) \vee \varphi_3) \vee \varphi_4)$. Ще считаме, че импликацията ($\Rightarrow$) и еквиваленцията ($\Leftrightarrow$) са с най-малък приоритет, конюнкцията ($\&$) и дизюнкцията ($\vee$) са със среден (и равен по между си) приоритет и унарните операции ($\neg$, $\forall$ и $\exists$) са с най-голям приоритет. Например всяка формула от вида $\varphi_1 \,\&\, \varphi_2 \Rightarrow \varphi_3 \vee \varphi_4$ е съкратен запис на формулата $((\varphi_1 \,\&\, \varphi_2) \Rightarrow (\varphi_3 \vee \varphi_4))$.

Дефиницията на формула е дадена по такъв начин, че да направи вярно следното твърдение:

2.4.14. Твърдение за еднозначен синтактичен разбор. *Всяка формула притежава единствено синтактично дърво.*

От еднозначността на синтактичния разбор следва например, че никоя формула не може да бъде едновременно от вида $(\varphi_1 \& \varphi_2)$ и $(\psi_1 \vee \psi_2)$, нито пък едновременно от вида $(\varphi_1 \Rightarrow \varphi_2)$ и $\forall \mathtt{x}\, \psi$. Също така от тук следва например, че една съждителна формула може да бъде едновременно от вида $(\varphi_1 \vee \varphi_2)$ и $(\psi_1 \vee \psi_2)$ само когато $\varphi_1 = \psi_1$ и $\varphi_2 = \psi_2$.

2.4.15. Пример. Синтактичното дърво на формулата

$$\forall \mathtt{x}\, (\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$

е илюстрирано във фигура 11.

✓ ***Задача* 17:** Кое е синтактичното дърво на следната формула:

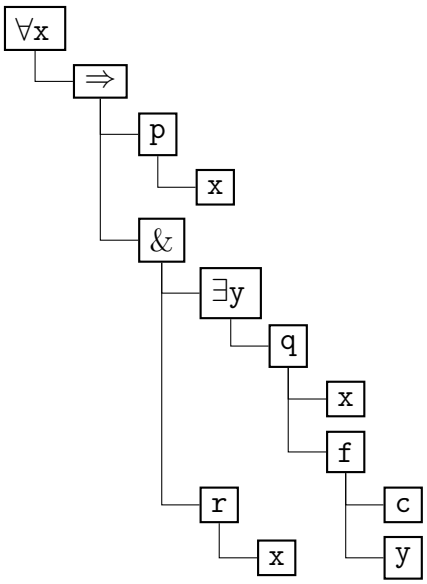
$$\exists \mathtt{x}\, (\forall \mathtt{x}\, \forall \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}))$$

Грешка ще бъде да считаме, че всяка формула има фиксиран смисъл. Да разгледаме например следната формула:

$$\mathtt{x} + 0 = \mathtt{x} \tag{5}$$

В тази формула $=$ е двуместен предикатен символ, $+$ е двуместен функционален символ, 0 е символ за константа и $\mathtt{x}$ е променлива. Какво ни казва тази формула? Например какви стойности може да приема променливата $\mathtt{x}$? Естествени числа, множества от думи или квадратни матрици 3×3? Какъв е смисълът на символа $+$? Събиране на естествени числа, обединение на множества или събиране на матрици?

По-нататък ще дефинираме понятието структура. Именно структурата ще дава конкретен смисъл на символите, които използваме във формулите. При различните структури тези символи ще придобиват



Фиг. 11. Синтактично дърво за формулата $\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$

различен смисъл и следователно е безсмислено да питаме за една формула дали е вярна, или не, без да сме казали коя е структурата, спрямо която интерпретираме символите във формулата. Тъй като смисълът на формулите зависи от структурата, спрямо която ги интерпретираме, то значи при някои структури една формула може да бъде вярна, а при други — не.

2.4.16. Пример. Да разгледаме няколко различни структури, в които да интерпретираме формулата $\mathtt{x} + 0 = \mathtt{x}$.

а) В първата структура, която ще разгледаме, нека възможните стойности на променливата $\mathtt{x}$ са естествените числа. Нека „=“ е равенство, символът за константа 0 е числото нула, а „+“ е събиране на естествени числа. В тази структура формулата $\mathtt{x}+0 = \mathtt{x}$ ни казва, че всяко естествено число, събрано с нула дава същото естествено число. Следователно в тази структура формулата е вярна.

б) Сега да разгледаме структура, която е същата като предходната, само че символът 0 се интерпретира като числото едно. В тази структура формулата ни казва, че всяко естествено число, събрано с единица, дава същото естествено число. Следователно в тази структура формулата не е вярна.

в) Да разгледаме структура, в която стойностите на променливите са езици (т.е. множества от думи), символът 0 се интерпретира като празният език $\varnothing$, а символът + като обединение на езици. В тази структура формулата ни казва, че ако обединим някой език с празния език, ще получим същия език. Следователно в тази структура формулата е вярна.

г) Да разгледаме структура, в която стойностите на променливите са реални числа или $+\infty$, символът 0 се интерпретира като $+\infty$, а + се интерпретира като операцията минимум. В тази структура формулата казва, че ако x е реално число или $+\infty$, то по-малкият елемент на множеството $\{x, +\infty\}$ е равен на x.

д) Да разгледаме структура, в която стойностите на променливите са многозначни функции от $\mathbb{N}$ в $\mathbb{N}$, символът 0 се интерпретира като празната функция, която никога не дава резултат, а + е обединението на многозначни функции. В тази структура формулата $\mathtt{x} + 0 = \mathtt{x}$ ни казва, че обединението на коя да е многозначна функция с функцията, която никога не дава резултат, е същата функция. От гледна точна на теория на изчислимостта това означава, че ако пуснем някой процес `x` паралелно с процес, който се зацикля и нищо не прави, това е все едно да пуснем единствено процеса `x` (очевидно това предполага математическата идеализация, че разполагаме с безкрайни ресурси и затова не се интересуваме, че зациклящият се процес може да забави пресмятанията).

2.5. Абстрактна алгебра и програмиране

Многообразия и квазимногообразия

В абстрактната алгебра се дефинират различни видове алгебрични структури, като също се формулират и аксиоми, които се удовлетворяват във всяка структура от съответния вид. Например аксиомите за поле са краен брой предикатни формули, които притежават следните две свойства:

– аксиомите за поле са верни във всяко поле;

– ако аксиомите за поле са верни в някоя структура, то тази структура е поле.

Ето една примерна формула, която обикновено се включва сред аксиомите за поле:

$$\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$$

Да забележим, че макар тази формула да е вярна във всяко поле, смисълът на функционалния символ + е различен в различните полета. Например в полето на реалните числа е вярно $2 + 2 = 4$, а в полето $\mathbb{Z}_3$ е вярно $2 + 2 = 1$.

2.5.1. Дефиниция. Сигнатура, в която единственият предикатен символ е двуместният предикатен символ „=“, се нарича *алгебрична сигнатура*. Да забележим, че при алгебричните сигнатури всички атомарни формули представляват равенства между термове.

2.5.2. Дефиниция. Нека $\mathfrak{sig}$ е алгебрична сигнатура. Казваме, че клас структури за сигнатурата $\mathfrak{sig}$ е *многообразие*,* ако съществува такова множество Γ от атомарни $\mathfrak{sig}$-формули, че за всяка структура за сигнатурата $\mathfrak{sig}$ е вярно, че:

– ако в структурата са верни формулите от Γ, то тази структура е елемент на многообразието;

– ако структурата е елемент на многообразието, то в нея са верни формулите от Γ.

Формулите от Γ се наричат *аксиоми* на многообразието.

Например групите и пръстените образуват многообразия, защото както при групите, така и при пръстените всяка от аксиомите представлява равенство между някакви изрази (т.е. термове). Едно малко по-общо понятие от многообразията са квазимногобразията. Няма широко известни класове от структури, които са квазимногообразия, но не са многообразия. Въпреки това квазимногообразията имат важни компютърни приложения.

2.5.3. Дефиниция. *Квазимногообразията* се дефинират аналогично на многообразията, само че при аксиомите позволяваме не само атомарни формули, но и формули от вида

$$\tau_1 = \sigma_1 \,\&\, \cdots \,\&\, \tau_n = \sigma_n \Rightarrow \tau = \sigma$$

Полетата са важен клас алгебрични структури, които не само че не са многобразие, но не са дори и квазимногообразие. Това е така, защото

*На английски *variety*. В алгебричната геометрия има алгебрични многообразия (algebraic varieties), които са различни от многообразията, за които говорим тук. Освен това в диференциалната геометрия на английски съществува терминът *manifold*, който няма нищо общо с variety, но на български също се превежда като „многообразие“.

една от аксиомите за поле казва, че всеки ненулев елемент има обратен елемент. С формула това може да бъде записано по следния начин:

$$\neg(\mathtt{x} = 0) \Rightarrow \mathtt{x}.\mathtt{x}^{-1} = 1$$

Използването на $\mathtt{x}^{-1}$ тук не е проблем, защото може да считаме, че $^{-1}$ е едноместен функционален символ.[*] Проблем е обаче наличието на отрицание от лявата страна на импликацията, защото това не се позволява при аксиомите на квазимногообразие. Може да се докаже, че този проблем е съществен — не е възможно така да измислим аксиомите за поле, че всяка от тях да има вида, позволен за аксиомите на квазимногообразие (вж. твърдение **??**).

Ще се спрем малко по-подробно на две многообразия, които имат разнообразни приложения в теоретичната информатика — моноидите и полупръстените.

Моноиди

Структура с една асоциативна операция (означена по-долу с „.“), която има неутрален елемент (означен по-долу с 1), се нарича *моноид*. С други думи, ако в някоя структура за сигнатурата $\langle 1, ., = \rangle$ са изпълнени следните аксиоми, то тя е моноид:

$$\mathtt{x}.(\mathtt{y}.\mathtt{z}) = (\mathtt{x}.\mathtt{y}).\mathtt{z}$$
$$\mathtt{x}.1 = \mathtt{x}$$
$$1.\mathtt{x} = \mathtt{x}$$

Пример. Някои примери за моноиди са:

а) Положителните цели числа с операцията умножение. Неутрален елемент е числото 1.

б) Положителните цели числа, като „.“ е операцията събиране, а символът 1 е числото нула. В тази структура аксиомата $\mathtt{x}.1 = \mathtt{x}$ казва, че всяко естествено число, събрано с нула, дава същото число.

в) Думите от дадена азбука Σ с операцията конкатенация. Неутрален елемент е празната дума ε. Този моноид се нарича *свободен моноид* над Σ.

г) Всички езици от думи над дадена азбука с операцията конкатенация на езици. Неутрален елемент е $\{\varepsilon\}$.

[*] Например може да считаме, че $\mathtt{x}^{-1}$ е по-четлив запис на терма $f(\mathtt{x})$.

д) Всички функции, изобразяващи елементите на дадено множество (например естествените числа) в същото множество, с операцията композиция на функции. Неутрален елемент е функцията идентитет.*

Асоциативни операции (а значи и моноиди) се появяват често в задачите, възникващи при програмиране. Благодарение на това много алгоритми могат да се разпаралелят по лесен и безопасен начин. Например ако е необходимо да пресметнем произведението

$$\prod_{i=1}^{8000} k_i$$

на компютър с осемядрен процесор, ние можем да извършим това по следния начин: на първото ядро пресмятаме $\prod i = 1^{1000} k_i$, на второто — $\prod_{i=1001}^{2000} k_i$, на третото — $\prod_{i=2001}^{3000} k_i$ и т. н.** Защо не можем да използваме този метод когато операцията не е асоциативна?

Полупръстени

Полупръстените образуват по-сложно многообразие. Полупръстенът е структура за сигнатурата $\langle 0, 1, +, ., = \rangle$, където символите „+“ и „.“ означават двуместни асоциативни операции, които имат неутрални елементи, означени съответно с 0 и 1. Операцията „+“ е комутативна. Освен това е в сила дистрибутивен закон за „+“ и „.“, а 0 действа като нулев елемент по отношение на операцията „.“. Ето как всичко това

*Да си припомним, че ако $f, g \colon M \to M$ са две функции, то тяхната композиция $f \circ g$ е функцията, за която

$$(f \circ g)(\mu) = f(g(\mu))$$

за всяко $\mu \in M$. Функцията идентитет id е функцията, за която

$$\mathrm{id}(\mu) = \mu$$

за всяко $\mu \in M$.

**Този метод няма да подобри крайното бързодействие, ако пресмятането на произведението може да се извърши бързо и на едно ядро. Ако има вероятност пресмятането на различните частични произведения да отнема различно по продължителност време, трябва да се обмисли използвато по-сложен метод за разпаралеляване, защото иначе може се случи седем от ядрата да си свършат бързо работата, а цялата трудна работа да се падне само на осмото ядро.

може да бъде изразено с формули:

$$\begin{aligned}
\mathtt{x} + (\mathtt{y} + \mathtt{z}) &= (\mathtt{x} + \mathtt{y}) + \mathtt{z} \\
\mathtt{x} + \mathtt{y} &= \mathtt{y} + \mathtt{x} \\
0 + \mathtt{x} &= \mathtt{x} \\
\mathtt{x}.(\mathtt{y}.\mathtt{z}) &= (\mathtt{x}.\mathtt{y}).\mathtt{z} \\
1.\mathtt{x} &= \mathtt{x} \\
\mathtt{x}.1 &= \mathtt{x} \\
\mathtt{x}.(\mathtt{y} + \mathtt{z}) &= (\mathtt{x}.\mathtt{y}) + (\mathtt{x}.\mathtt{z}) \\
(\mathtt{y} + \mathtt{z}).\mathtt{x} &= (\mathtt{y}.\mathtt{x}) + (\mathtt{z}.\mathtt{x}) \\
0.\mathtt{x} &= 0 \\
\mathtt{x}.0 &= 0
\end{aligned}$$

Пример. Някои примери за полупръстени са:

а) Естествените числа с операции събиране и умножение.

б) Езиците над дадена азбука като операцията „+“ е обединението на езици, операцията „.“ е конкатенацията на езици, 0 е празният език, а 1 е $\{\varepsilon\}$.

в) Многозначните изчислими функции. Операцията „+“ е обединението на две многозначни функции, операцията „.“ е композицията на многозначни функции, 0 е функцията която никога не връща резултат, а 1 е функцията идентитет. Този полупръстен се използва в теория на изчислимостта и може да се използва за моделиране на алгоритмични пресмятания, при които паралелните процеси не комуникират по между си. [11]*

г) Реалните числа заедно с $+\infty$, където дефинираме $\mathtt{x} + \mathtt{y}$ да бъде по-малкото от $\mathtt{x}$ и $\mathtt{y}$, а $\mathtt{x}.\mathtt{y}$ е сборът на $\mathtt{x}$ и $\mathtt{y}$. Неутрален елемент за „+“ е $+\infty$ и неутрален елемент за „.“ е 0. Този полупръстен се нарича *тропически полупръстен*.** Той има различни прило-

*За съжаление все още не е открит и може би не съществува хубав математически формализъм, който може да се използва за моделиране на комуникиращи паралелни процеси. Такива процеси може да се моделират например посредством *π-смятането* и *джойн-смятането*, но и двете имат много лоши алгебрични свойства.

**Приложната математика обикновено използва математически обекти, които са били дефинирани от математици теоретици, които изобщо не са се интересували, че с измислените от тях понятия в бъдеще ще може да се решават практически задачи. Тропическият полупръстен е рядък пример за интересен математически обект, който е бил откриван и преоткриван много пъти от математици приложници и едва в последствие е заинтересувал и математиците теоретици.

жения, едно от които е свързано с анализ на дискретни системи, т.е. системи, чието поведение не е „гладко“ и затова не може да се опише посредством диференциални уравнения. Примери за дискретни системи са транспортните мрежи, комуникационните и компютърните мрежи, централните процесори на компютрите, производствените цехове в заводите и др.

Многосортни структури

Моноидите и полупръстените представляват примери за т. н. едносортни структури. При едносортните структури всички индивиди са от един и същи сорт — в случая това са елементите на моноида или полупръстена. Има обаче и *многосортни структури*. Например линейните пространства са пример за двусортна структира, в която има два вида обекти — скалари и вектори. За всяка променлива, срещаща се във формула от многосортен език е определен сорт, който показва кои са позволените стойности на съответната променлива. Например една от аксиомите за линейно пространство е двусортната формула

$$(\mathtt{x}.\mathtt{y}).\mathtt{v} = \mathtt{x}.(\mathtt{y}.\mathtt{v})$$

в която стойностите на `x` и `y` са всевъзможните скалари в структурата (напр. реални числа), а стойностите на `v` — всевъзможните вектори в структурата.

Въпреки че теорията на многосортните структури не е по-сложна от тази на едносортните, с цел да избегнем някои технически усложнения в този курс използваме само едносортни структури.

Алгебрично програмиране

Има една група езици за формална спецификация и програмиране, при които дефинираме различните типове данни аксиоматично — посредством задаване на свойствата, удовлетворявани от обектите от съответните типове. По практически съображения се налага да наложим някои ограничения върху вида на аксиомите. Тези ограничения са такива, че всъщност алгебричното програмиране се основава на използването на многосортни квазимногообразия. Мауде* е един от популярните езици за алгебрично програмиране. Ето как изглежда на мауде описанието на квазимногообразието на моноидите:

*`http://maude.cs.uiuc.edu`

```
fth MONOID is
   sort Elt .
   op 1 : → Elt .
   op _*_ : Elt Elt → Elt [assoc id: 1] .
endfth
```

Тук дефинираме *сорт* `Elt` и две операции `1` и `*`. Операцията `1` е просто елемент на моноида, а операцията `*` е двуместна, асоциативна и притежаваща `1` като неутрален (единичен) елемент.

Полупръстените могат да бъдат дефинирани на мауде по следния начин:

```
fth SEMIRING is
   sort Elt .
   op 0 : → Elt .
   op _+_ : Elt Elt → Elt [assoc comm id: 0] .
   op 1 : → Elt .
   op _*_ : Elt Elt → Elt [assoc id: 1] .
   vars x y z : Elt .
   eq x * (y + z) = (x * y) + (x * z) [nonexec] .
   eq (x + y) * z = (x * z) + (y * z) [nonexec] .
   eq 0 * x = 0 [nonexec] .
   eq x * 0 = 0 [nonexec] .
endfth
```

Тук елементът 0 е описан като неутрален за асоциативната и комутативната операция +, елементът 1 като неутрален за асоциативната *, а дистрибутивните закони са описани с явно зададени аксиоми. Атрибутът `nonexec` казва на мауде, че не е нужно да обръща внимание на последните четири аксиоми докато пресмята стойността на операциите + и *.

От математическа гледна точка езиците за алгебрично програмиране се основават на съществуването на т. н. *инициални структури*, които са единствени с точност до изоморфизъм. Съществуването на инициална структура за всяко многообразие е доказано от Гарет Биркхоф [5], а за квазимногообразията — от Анатолий Иванович Мальцев [23, 24, стр. 271].

Естествено възниква въпросът каква част от структурите, които бихме искали да използваме при програмирането, могат да се опишат като инициални структури на някое квазимногообразие. Йоханес Бергстра и Джон Тъкер са доказали [3, 4], че ако е възможно операциите на

една структура да се пресмятат алгоритмично,[*] то тогава тази структура може да се опише аксиоматично като инициална структура на многообразие, стига да обогатим структурата с краен брой нови операции. Следователно всички структури, чиито операции могат да се пресмятат с компютър, могат да се представят като инициални структури на някакво многообразие.

Въпреки че езиците за алгебрично програмиране наистина могат да се използват за програмиране, тяхното предназначение е друго — като езици за формална спецификация и описание на семантиката на езици за програмиране. Има различни езици за спецификация, но едно ценно свойство, което отличава езиците за алгебрично програмиране от повечето други езици за спецификация, е това, че при алгебричните езици самата спецификация ни дава изпълним код. В някои случаи скоростта на изпълнение на такава програма се оказва изненадващо бърза. Например ако запишем на мауде денотационната семантика на езика скийм, то ще получим интерпретатор на скийм, чиято скорост достига до 75% от скоростта на обикновен, наивно написан интепретатор на скийм. И въпреки че ако оставим на мауде да изпълни формалната спецификация на виртуалната машина на джава, ще получим доста бавен интерпретатор, да правим това изобщо не е безсмислено. Макар и бавен, този интерпретатор със сигурност отговаря на формалната си спецификация (т. е. не съдържа програмни грешки). Затова такъв интерпретатор може да се използва за безопасно изпълнение на програми, които са потенциално злонамерени (вируси и др.).[**]

[*] Такива структури се наричат *изчислими* или *рекурсивни*.

[**] Всъщност когато става въпрос за софтуерна безопасност рядко можем да имаме пълна сигурност за каквото и да е. Например злонамереният код на джава може се възползва от програмни грешки в интерпретатора на мауде, за да излезе от наложените му ограничения. Ако пък целта на злонамерения код е не да модифицира потребителските данни, а само шпионска, тогава защитата става още по-сложна. Дори и без да излиза от наложените му ограничения, такъв код може да осъществи нерегламентирано предаване на данни, като влияе по подходящ начин на натоварването на централния процесор — натоварване, което може да бъде следено от разстояние или посредством електромагнитните излъчвания на процесора, или по шума, издаван от вентилатора, или дори по консумацията на електроенергия от електрическата мрежа.

2.6. Свободни и свързани променливи

Уводни бележки

Променливите в математиката и програмирането са два вида — свободни и свързани. Двата вида променливи са много различни един от друг. Да разгледаме израза

$$\sum_{i=1}^{n} i^2$$

За да пресметнем този израз е нужно да знаем каква е стойността на променливата n, но не и стойността на променливата i. Въпреки че в този израз се срещат две променливи — n и i – очевидно тези променливи са много различни. Ако искаме с горния израз да дефинираме функция, тази функция ще зависи само от n, но не и от i:

$$f(n) = \sum_{i=1}^{n} i^2$$

Променливите, подобни на n, се наричат *свободни променливи*, а подобните на i се наричат *свързани променливи*.

Ето още няколко примера. В следващия израз променливата x е свързана, а y е свободна:

$$\int_0^1 f(x,y)dx$$

В следния програмен блок променливата `i` е свързана, а променливите `a` и `x` са свободни:

```
{
   int i;
   for(i=1;i<1000;i++){
      x[i] = (x[i-1]*x[i-1]) % a;
   }
}
```

И така, да обобщим — стойността (смисъла) на един израз зависи от стойността на свободните променливи, срещащи се в него, но не зависи от стойността на свързаните променливи в израза.

Едно друго различие между свободните и свързаните променливи е следното. Ако в един израз заменим всяко срещане на свободната променлива x с някакъв израз, ще се получи пак смислен израз. Да заменяме свързана променлива с израз не е позволено.

Например ако в израза

$$\sum_{i=1}^{n} i^2$$

заменим n с k^2+5 получаваме израза

$$\sum_{i=1}^{k^2+5} i^2$$

Ако в израза

$$\int_0^1 f(x,y)dx$$

заменим y с z^2+5 получаваме израза

$$\int_0^1 f(x,z^2+5)dx$$

Ако в горния програмен блок заменим променливата `a` с израза `a*a+5` получаваме новия програмен блок:*

```
{
   int i;
   for(i=1;i<1000;i++){
      x[i] = (x[i-1]*x[i-1]) % (a*a+5);
   }
}
```

Ако по подобен начин заменяме свързаните променливи с изрази, обикновено се получават безсмислици. Например ако в израза

$$\sum_{i=1}^{n} i^2$$

заменим i с k^2+5 получаваме

$$\sum_{(k^2+5)=1}^{n} (k^2+5)^2$$

*При подобни замени трябва да се спазват типовете. Например ако променливата `a` се срещаше отляво на оператор за призвояване, тогава бихме могли да заменяме тази променлива само с израз, притежаващ определен адрес в паметта. В този случай замяната на `a` с `a*a+5` не би била коректна, но замяната с `x[4]` ще бъде коректна.

Ако в израза

$$\int_0^1 f(x,y)dx$$

заменим x с z^2+5 получаваме

$$\int_0^1 f(z^2+5,y)d(z^2+5)$$

Ако в горния програмен блок заменим `i` с `a*a+5` получаваме

```
{
   int a*a+5;
   for(a*a+5=1;a*a+5<1000;(a*a+5)++){
      x[i] = (x[i-1]*x[i-1]) % (a*a+5);
   }
}
```

Да забележим, че във всеки от горните случаи свързаната променлива има нещо като „свързващ оператор“, от който съвсем ясно личи, че се получава безсмислица. Това са $\sum_{(k^2+5)=1}^{n}$, $\int_0^1 \ldots d(z^2+5)$ и декларацията `int a*a+5;`.

При преименуване на свързана променлива смисълът на израза се запазва. Например

$$\sum_{i=1}^{n} i^2 = \sum_{j=1}^{n} j^2 \neq \sum_{i=1}^{m} i^2$$

$$\int_0^1 f(x,y)dx = \int_0^1 f(z,y)dz \neq \int_0^1 f(x,z)dx$$

Също и в горния програмен блок смисълът се запазва, ако преименуваме `i` на `j`, но не и ако преименуваме `a` на `b`.

Една друга разлика между свободните и свързаните променливи е това, че свободните променливи имат глобална видимост, а свързаните — локална. Да разгледаме следния програмен фрагмент:

```
{
   int i, j;
   i = a * a;
   {
```

свободни променливи	*свързани променливи*
изразът зависи от стойността им	стойността им не е релевантна
може да се заместват с изрази	не може да се заместват
не може да се преименуват	може да се преименуват
глобална видимост	локална видимост

Таблица 3. Свойства на свободните и свързаните променливи

```
      int i;
      i = b + 5;
      j = 3;
   }
   {
      int i, a;
      i = b + j;
      a = i * i;
      b = a * i;
   }
}
```

В този фрагмент са декларирани три променливи с име `i`. Въпреки че използват едно и също име, тези три променливи са напълно различни една от друга. Освен това всяка си има своя област на видимост. Променливите `i`, декларирани във вътрешните блокове, са видими само в рамките на тези блокове. Променливата `i`, декларирана във външния блок, също е видима само в рамките на този блок, но не и извън него. Освен това вътрешните променливи `i` скриват външната променлива `i`, в следствие на което тя не се вижда във вътрешните блокове. Променливата `j` обаче не се скрива от вътрешна декларация и затова се вижда и във вътрешните блокове.

В този програмен фрагмент се използват и две свободни променливи — `a` и `b`. Всяко срещане на променлива `b` в този програмен фрагмент се отнася за една и съща променлива. Във втория вътрешен блок обаче е декларирана свързана променлива `a`. Тази свързана променлива `a` скрива свободната променлива `a`.

При свързаните променливи може да има различни свързани променливи с едно и също име, но при свободните това е невъзможно — едноименните свободни променливи винаги са една и съща променлива.

Казаното дотук за свободните и свързаните променливи е резюмирано в таблица 3.

Всички променливи, които се срещат в един терм са свободни, а не свързани. За да се убедим в това, да разгледаме едно по едно свойствата от таблица 3:

– От разгледаните разнообразни примери за алгебрични структури се вижда, че в общия случай стойността на един терм зависи от стойността на променливите, които се срещат в него. Например стойността на терма $\mathtt{x}+\mathtt{x}$ зависи от стойността на променливата $\mathtt{x}$.

– Ако заменим една променлива в терм с друг терм, пак получаваме терм. Например ако заменим променливата $\mathtt{x}$ в терма $\mathtt{f}(\mathtt{x})$ с терма $\mathtt{g}(\mathtt{x},\mathtt{y})$, ще получим терма $\mathtt{f}(\mathtt{g}(\mathtt{x},\mathtt{y}))$.

– Ако преименуваме променливите в един терм, получаваме различен терм. Термовете $\mathtt{f}(\mathtt{x})$ и $\mathtt{f}(\mathtt{y})$ имат различен смисъл.

– Всички променливи $\mathtt{x}$, които се срещат в един терм, означават една и съща променлива $\mathtt{x}$. Следователно променливите в термовете имат глобална видимост.

Всички променливи, които се срещат в една атомарна формула, също са свободни, а не свързани. В това можем да се убедим по същия начин:

– Не е безсмислено да се питаме каква е верността на една атомарна формула при някакви конкретни стойности на променливите, които се срещат в нея. Например верността на атомарната формула $\mathtt{x} = 0$ зависи от стойността на променливата $\mathtt{x}$.

– Ако заменим една променлива в атомарна формула с друг терм, пак получаваме атомарна формула. Например ако заменим променливата $\mathtt{x}$ в атомарната формула $\mathtt{p}(\mathtt{x})$ с терма $\mathtt{g}(\mathtt{x},\mathtt{y})$, ще получим атомарната формула $\mathtt{p}(\mathtt{g}(\mathtt{x},\mathtt{y}))$.

– Ако преименуваме променливите в една атомарна формула, получаваме различна атомарна формула. Атомарните формули $\mathtt{p}(\mathtt{x})$ и $\mathtt{p}(\mathtt{y})$ имат различен смисъл.

– Всички променливи $\mathtt{x}$, които се срещат в една атомарна формула, означават една и съща променлива $\mathtt{x}$. Следователно променливите в атомарните формули имат глобална видимост.

В една произволна формула обаче може да има и свързани променливи. Това се дължи на кванторите. Да разгледаме например формулата $\forall\mathtt{x}\,\mathtt{p}(\mathtt{x})$. В нея променливата $\mathtt{x}$ е свързана, защото:

– Верността на формулата $\forall\mathtt{x}\,\mathtt{p}(\mathtt{x})$ не зависи от никаква конкретна стойност на променливата $\mathtt{x}$.

– Ако заменим в тази формула променливата $\mathtt{x}$ с терма $\mathtt{f(x)}$ получаваме безсмислица: $\forall \mathtt{f(x)}\, \mathtt{p(f(x))}$.
– Ако преименуваме в тази формула променливата $\mathtt{x}$ с $\mathtt{y}$, получаваме формула с напълно същия смисъл: $\forall \mathtt{y}\, \mathtt{p(y)}$.
– Кванторите създават локална видимост за променливата си. Ако в контекста на формулата $\forall \mathtt{x}\, \mathtt{p(x)}$ се срещат някакви променливи $\mathtt{x}$, то това са напълно различни променливи $\mathtt{x}$. Например първата променлива $\mathtt{x}$ от формулата $\mathtt{q(x)} \,\&\, \forall \mathtt{x}\, \mathtt{p(x)}$ е напълно различна променлива $\mathtt{x}$ (която впрочем е свободна), от втората и третата променлвива $\mathtt{x}$, която е свързана.

Подформули

За да направим по-ясно кои са свързаните и кои са свободните променливи във формула, ще въведем едно помощно понятие:

✓ **2.6.1. Дефиниция.** Когато формулата ψ е част от формулата φ (т.е. ψ е подниз на φ), казваме, че ψ е *подформула* на φ.

✓ **2.6.2.** За да разберем правилно смисъла на току-що дадената дефиниция, е най-добре да си мислим не за самите формули, които са някакви редици от символи, а за техните синтактични дървета. Ако си мислим за синтактичното дърво на една формула, някои от поддърветата ѝ ще бъдат нейни подформули, а други (по-малките) — термове. Подформулите на една формула се получават от всички поддървета, които са формули, а не термове. Областта на действие на всеки квантор е поддървото, което се намира под съответния квантор. Разгледайте фигура 12, в която е илюстрирано синтактичното дърво на формулата

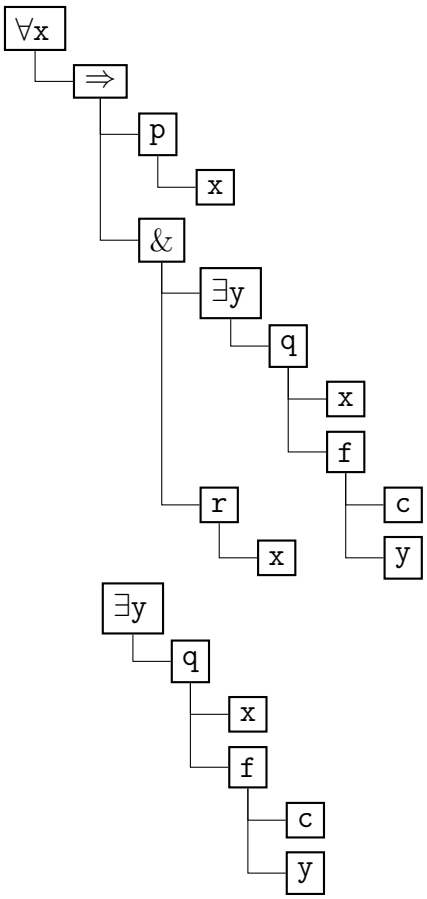
$$\forall \mathtt{x}\, (\mathtt{p(x)} \Rightarrow \exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r(x)})$$

както и синтактичното дърво на областта на действие на квантора $\exists \mathtt{y}$ в тази формула.

Ако решим да си мислим не за синтактичните дървета, а за самите формули като редици от символи, тогава дефиниция 2.6.1 ще бъде вярна само тогава, когато изписваме формулата изцяло, без да пропускаме нито едни скоби. Да разгледаме отново формулата

$$\forall \mathtt{x}\, (\mathtt{p(x)} \Rightarrow \exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r(x)})$$

Според дефиниция 2.6.1, ако една подформула започва с квантор, тогава тази подформула е областта на действие на този квантор. Една



Фиг. 12. Синтактично дърво на формула и подформула

подформула на горната формула е формулата $\exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x})$. Тази формула започва с квантора $\exists \mathtt{y}$ и значи излиза, че тя трябва да е областта на действие на този квантор. Достатъчно е обаче да погледнем дървото от фигура 12, за да видим, че това е невярно!

Тази грешка се обяснява по следния начин — не е вярно, че формулата $\exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x})$ започва с квантор. Ако я напишем изцяло, без да пропускаме скоби, тогава ще видим че тази формула започва не с квантор, а със скоба, защото дефиниция 2.4.10 в) изисква около всяка конюнкция да се слагат скоби:

$$(\exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$

Следователно тази подформула започва не с квантор, а със скоба.

Тъй като е много неудобно всеки път да преценяваме къде се слагат всички нужни скоби и дали дадена подформула започва с квантор,

или със скоба, която не е написана, най-добре е когато откриваме подформулите на една формула, да си мислим за синтактичното дърво на формулата, а не за самата формула.

✓ ***Задача* 18:** Намерете всички подформули, на формулата от фигура 12.

Свързани и свободни променливи във формула

Ако приложим всичко казано до момента за свободните и свързаните променливи в една формула, ще получим следната дефиниция:

✓ **2.6.3. Дефиниция.**

а) Когато формулата φ съдържа подформула от вида $\forall \mathtt{x}\, \psi$ или $\exists \mathtt{x}\, \psi$ (където ψ е формула), казваме, че тази подформула е *областта на действие* на квантора $\forall \mathtt{x}$ или $\exists \mathtt{x}$, с който започва подформулата.

б) Едно срещане на променливата $\mathtt{x}$ в някоя формула е *свързано*, ако попада в областта на действие на някой квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$.

в) Възможно е едно срещане на променливата $\mathtt{x}$ в някоя формула да попада едновременно в областта на действие на няколко квантора $\forall \mathtt{x}$ или $\exists \mathtt{x}$. Измежду всички тези квантори, за квантора, чиято област на действие е най-малка,* казваме, че *управлява* това срещане на променливата $\mathtt{x}$.

г) Едно срещане на променливата $\mathtt{x}$ в някоя формула е *свободно*, ако не попада в областта на действие на никой квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$.

д) Променливата $\mathtt{x}$ е *свободна променлива* на формулата φ, ако $\mathtt{x}$ се среща в φ на място, което не попада в областта на действие на никой квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$. С други думи $\mathtt{x}$ е свободна променлива, ако $\mathtt{x}$ има свободно срещане във формулата.

Много е важно да се разберат следващите три примера.

✓ **2.6.4. Пример.** В безкванторните формули няма квантори, следователно не е възможно една променлива да попадне в областта на действие на квантор. Следователно всички променливи, които се срещат в безкванторните формули са техни свободни променливи. Например променливите $\mathtt{x}$, $\mathtt{y}$ и $\mathtt{z}$ са свободните променливи на формулата

$$(\mathtt{p}(\mathtt{x}, \mathtt{c}) \vee \neg \mathtt{q}(\mathtt{y})) \Rightarrow \mathtt{p}(\mathtt{z}, \mathtt{x})$$

*Тъй като всички тези подформули се включват една в друга, е все едно как ще уточним смисъла на думата „най-малка“: най-малка по дължина или най-малка по включване.

Атомарните формули са вид безкванторни формули, следователно всички променливи, които се срещат в атомарните формули са техни свободни променливи.

✓ **2.6.5. Пример.** Променливите $\mathtt{x}$ е единствената свободна променлива на формулата

$$\forall \mathtt{x}\, \forall \mathtt{y}\, (\mathtt{p}(\mathtt{x}, \mathtt{c}) \vee \neg \mathtt{q}(\mathtt{y})) \Rightarrow \forall \mathtt{z}\, \mathtt{p}(\mathtt{z}, \mathtt{x})$$

В тази формула променливата $\mathtt{x}$ от подформулата $\mathtt{p}(\mathtt{x}, \mathtt{c})$ е свързана, защото попада в областта на действие на квантора $\forall \mathtt{x}$. Същевременно променливата $\mathtt{x}$ от подформулата $\mathtt{p}(\mathtt{z}, \mathtt{x})$ е свободна, защото не попада в областта на действие на квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$.

Също както в езиците за програмиране декларирането на локална променлива в даден програмен блок скрива едноименните променливи, декларирани в по-външен блок, така и кванторите скриват едноименните свободни променливи, както и свързаните променливи с по-външен квантор.

✓ **2.6.6. Пример.** Във формулата

$$\forall \mathtt{x}\, (\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}) \vee \mathtt{p}(\mathtt{x})) \vee \mathtt{p}(\mathtt{x})$$

в първата атомарна формула $\mathtt{p}(\mathtt{x})$ променливата $\mathtt{x}$ се управлява от втория квантор, във втората атомарна формула $\mathtt{p}(\mathtt{x})$ променливата $\mathtt{x}$ се управлява от първия квантор, а в третата — променливата $\mathtt{x}$ е свободна.

✓ ***Задача* 19:** За всяка променлива, срещаща се във формулата

$$\forall \mathtt{x}\, (\forall \mathtt{x}\, \exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{y}))$$

определете дали е свободна или свързана. Кои са кванторите, управляващи свързаните променливи? Коя е областта на действие на всеки един от кванторите? Кое е множеството от свободните променливи на тази формула?

Преименуване на свързаните променливи

В уводните бележки на този раздел бе споменато, че свързаните променливи в един израз може да се преименуват без това да промени смисъла на израза. Ако формулата φ може да се сведе до формулата ψ посредством преименуване на свързаните променливи, казваме, че тези две формули са *конгруентни* (точна дефиниция на това понятие е дадена малко по-долу). Ако две формули са конгруентни, то техният смисъл е един и същ.

2.6.7. Пример. Формулите $\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y})$ и $\forall \mathtt{z}\, \mathtt{p}(\mathtt{z}, \mathtt{y})$ са конгруентни. Втората формула може да се получи от първата като преименуваме $\mathtt{x}$ на $\mathtt{z}$. Тези формули обаче не са конгруентн с формулата $\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{t})$, защото променливата $\mathtt{y}$ е свободна и не може да се преименува.

Ако във формулата $\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y})$ преименуваме свързаната променлива $\mathtt{x}$ на $\mathtt{y}$, получаваме формулата $\forall \mathtt{y}\, \mathtt{p}(\mathtt{y}, \mathtt{y})$. Тези две формули обаче не са конгруентни, защото новопоявилият се квантор $\forall \mathtt{y}$ обхваща и свободната променлива $\mathtt{y}$ и я превръща от свободна в свързана.

Можем да дефинираме конгруентните формули по следния начин:

✓ **2.6.8. Дефиниция.** Две формули φ и ψ са *конгруентни*, ако са изпълнени следните условия:

а) двете формули имат една и съща дължина;

б) ако на дадена позиция в едната формула има символ, който не е променлива, то в другата формула стои същият символ;

в) ако на дадена позиция в едната формула има променлива и тя е свободна променлива, то на същата позиция в другата формула стои същата променлива и тя отново е свободна променлива;

г) ако на дадена позиция в едната формула има променлива и тя е свързана променлива, то на същата позиция в другата формула също стои променлива и тя отново е свързана променлива. Управляващите квантори на тези две променливи стоят на една и съща позиция.

✓ ***Задача* 20:** Конгруентни ли са формулите:

а) $\forall \mathtt{x}\, \forall \mathtt{y}\, \forall \mathtt{y}\, \mathtt{p}(\mathtt{x}, \mathtt{y})$ и $\forall \mathtt{x}\, \forall \mathtt{z}\, \forall \mathtt{y}\, \mathtt{p}(\mathtt{x}, \mathtt{y})$

б) $\forall \mathtt{x}\, \forall \mathtt{y}\, \forall \mathtt{y}\, \mathtt{p}(\mathtt{x}, \mathtt{y})$ и $\forall \mathtt{y}\, \forall \mathtt{x}\, \forall \mathtt{x}\, \mathtt{p}(\mathtt{y}, \mathtt{x})$

в) $\forall \mathtt{x}\, \forall \mathtt{y}\, \forall \mathtt{y}\, \mathtt{p}(\mathtt{x}, \mathtt{y})$ и $\forall \mathtt{y}\, \forall \mathtt{y}\, \forall \mathtt{x}\, \mathtt{p}(\mathtt{y}, \mathtt{x})$

След като разгледаме внимателно условията в дефиницията за конгруентност, може да забележим, че тя дефинира рефлексивна, симетрична и транзитивна релация. Следователно конгруентността е релация на еквивалентност и е вярно следното следствие:

✓ **2.6.9. Следствие.**

а) $\varphi \equiv \varphi$

б) ако $\varphi \equiv \psi$, *то* $\psi \equiv \varphi$

в) ако $\varphi \equiv \psi$ *и* $\psi \equiv \chi$, *то* $\varphi \equiv \chi$

Вече споменахме, че ще искаме така да разработваме теорията на предикатната логика, че да можем да отъждествяваме конгруентните формули, т.е. да работим „с точност до конгруентност“. Това означава, че трябва да бъдат верни твърдения, подобни на следващото.

2.6.10. Твърдение. *Ако две формули са конгруентни, то те имат едни и същи свободни променливи.*

Доказателство. Дефиницията на конгруентност ни гарантира, че ако на дадена позиция в едната формула има свободна променлива, то на същата позиция в другата формула стои същата променлива и тя също е свободна. ∎

За да не се усложняваме ненужно, така формулирахме дефиницията за формула (2.4.10), че да можем да пишем „объркващи“ формули като

$$\forall \mathtt{x}\, \forall \mathtt{y}\, (\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \vee \mathtt{q}(\mathtt{x}, \mathtt{y}))$$

В тази формула има два квантора с променливата $\mathtt{x}$. Променливата $\mathtt{x}$ в подформулата $\mathtt{p}(\mathtt{x}, \mathtt{y})$ ще бъде управлявана от втория квантор $\forall \mathtt{x}$, а променливата $\mathtt{x}$ в подформулата $\mathtt{q}(\mathtt{x}, \mathtt{y})$ — от първия квантор $\forall \mathtt{x}$. Но въпреки, че използването на такива формули се позволява от дефинициите, обикновено е добре да пишем „нормални“ и по-лесни за осмисляне формули, в които няма квантори с една и съща променлива. Това винаги е възможно, защото ако има квантори с повтарящи се променливи, с подходящо преименуване на свързаните променливи може да получим конгруентна формула, в която няма квантори с една и съща променлива.

Друг вид „обърквща“ формула е например следната:

$$\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}) \Rightarrow \mathtt{q}(\mathtt{x})$$

И тази формула е обърквща, защото в нея променливата $\mathtt{x}$ се среща хем като свободна, хем като свързана. В подформулата $\mathtt{p}(\mathtt{x})$ тя е свързана, а в $\mathtt{q}(\mathtt{x})$ — свободна. Но и тук няма проблем да се освободим от тази странност като преименуваме свързаната променлива $\mathtt{x}$ на някоя друга.

Ако пък Θ е някакво (крайно) множество от променливи, които по някакви причини „не харесване“, то отново — няма да има проблеми така да преименуваме, че да намерим конгруентна формула, в която няма да има квантори с променливи от Θ. Следващата лема доказва всички тези неща накуп.

✓ **2.6.11. Лема за преименуване на свързаните променливи.** *За всяка формула φ и крайно множество Θ от променливи можем да намерим конгруентна на φ формула, в която не се срещат квантори с променливи от Θ, всички квантори са с различни променливи и никоя от променлива не се среща едновременно като свързана и свободна.*

Доказателство. Нека Θ' е обединението на Θ с множеството от всички променливи (свързани и несвързани), които се срещат в φ. Множеството Θ' е крайно.

Ще построим редица от конгруентни формули $\varphi = \varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n$, в която последната формула ще отговаря на изискванията на лемата — всички кванторни променливи ще са различни и никоя кванторна променлива няма да е елемент на Θ'.

Ако разполагаме с формулата φ_i, можем да получим формулата φ_{i+1} по следния начин. Нека $\forall \mathtt{z}\, \psi$ или $\exists \mathtt{z}\, \psi$ е произволна подформула на φ_i, чиято променлива $\mathtt{z}$ е елемент на Θ'. Нека $\mathtt{z}'$ е произволна променлива, която не се среща нито в φ_i, нито е елемент на Θ'. Да преименуваме всички управлявани от квантора $\forall \mathtt{z}$ или $\exists \mathtt{z}$ променливи (в това число и самата кванторна променлива) от $\mathtt{z}$ на $\mathtt{z}'$. Нека така получената конгруентна формула бъде φ_{i+1}.

На всяка стъпка броят на кванторите с променливи от Θ' намалява и значи процесът на строене на тази редица от конгруентни формули спира. Това ще се случи тогава, когато стигнем до формула φ_n, в която няма повече квантори с променливи от Θ'. Тъй като включихме в Θ' всички първоначални променливи на формулата, то значи всяка кванторна променлива е била преименувана. При това преименуване обаче винаги избирахме и затова φ_n не съдържа два квантора с една и съща променлива, нито пък променлива, която е едновременно свързана и свободна. ∎

✓ **2.6.12. Пример.** Да видим с конкретен пример как работи току-що доказаната лема. Нека

$$\varphi = \forall \mathtt{y}\, \forall \mathtt{x}\, (\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \vee \mathtt{p}(\mathtt{x}, \mathtt{y})) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

и $\Theta = \{\mathtt{y}, \mathtt{z}\}$. В тази формула има много „дефекти“ — квантори с една и съща променлива ($\mathtt{x}$), променлива, която е хем свързана, хем свободна ($\mathtt{y}$), както и квантори с променливи от Θ. Нека поправим тези дефекти по начина, описан в доказателството на лемата. Нека най-напред изберем втория квантор $\forall \mathtt{x}$. Ще заменим променливата му $\mathtt{x}$ с $\mathtt{z}_{13}$ (избирама коя да е променлива, която не се среща нито във формулата,

нито е елемент на Θ). След замяната получаваме

$$\forall \mathtt{y}\, \forall \mathtt{x}\, (\forall \mathtt{z}_{13}\, \mathtt{p}(\mathtt{z}_{13}, \mathtt{y}) \vee \mathtt{p}(\mathtt{x}, \mathtt{y})) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

Сега да изберем другия квантор $\forall \mathtt{x}$. Ще заменим $\mathtt{x}$ например с $\mathtt{y}_{42}$. Получаваме

$$\forall \mathtt{y}\, \forall \mathtt{y}_{42}\, (\forall \mathtt{z}_{13}\, \mathtt{p}(\mathtt{z}_{13}, \mathtt{y}) \vee \mathtt{p}(\mathtt{y}_{42}, \mathtt{y})) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

Накрая да изберем $\forall \mathtt{y}$ и да заменим $\mathtt{y}$ с $\mathtt{z}'''$. Получаваме

$$\forall \mathtt{z}'''\, \forall \mathtt{y}_{42}\, (\forall \mathtt{z}_{13}\, \mathtt{p}(\mathtt{z}_{13}, \mathtt{z}''') \vee \mathtt{p}(\mathtt{y}_{42}, \mathtt{z}''')) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

Така получената формула нито съдържа квантори с една и съща променлива, нито има променлива, която е хем свързана, хем свободна, нито има квантори с променливи от Θ. Същевременно тази формула е конгруентна с φ, защото се получава с преименувания на свързаните променливи.

✓ ***Задача* 21:** Намерете формула, конгруентна на

$$\forall \mathtt{x}\, \forall \mathtt{y}\, (\exists \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \,\&\, \mathtt{q}(\mathtt{x}, \mathtt{z}) \Rightarrow \exists \mathtt{z}\, \mathtt{p}(\mathtt{y}, \mathtt{z}))$$

която не съдържа квантори с повтарящи се променливи, никоя променлива не се среща едновременно като свързана и свободна и променливите $\mathtt{x}$ и $\mathtt{y}$ не се срещат.

2.6.13. Лема. *а) Ако* $\forall \mathtt{x}\, \varphi \equiv \forall \mathtt{x}\, \psi$, *то* $\varphi \equiv \psi$.

б) Ако $\exists \mathtt{x}\, \varphi \equiv \exists \mathtt{x}\, \psi$, *то* $\varphi \equiv \psi$.

Доказателство. Двете условия на тази лема се доказват по един и същ начин. Ще докажем първото.

Щом $\forall \mathtt{x}\, \varphi$ и $\forall \mathtt{x}\, \psi$ са с една и съща дължина, то значи и φ и ψ са с една и съща дължина. Щом символите, които не са променливи, в $\forall \mathtt{x}\, \varphi$ и $\forall \mathtt{x}\, \psi$ са едни и същи, то също и в φ и ψ тези символи са едни и същи.

Ако $\mathtt{z}$ е свободна променлива в φ, то това срещане на $\mathtt{z}$ или е свободна и в $\forall \mathtt{x}\, \varphi$, или $\mathtt{z} = \mathtt{x}$, в който случай началният квантор $\mathtt{x}$ ще управлява $\mathtt{z}$. В първия случай от конгруентността на формулите $\forall \mathtt{x}\, \varphi$ и $\forall \mathtt{x}\, \psi$ получаваме, че на същата позиция във втората формула също стои променливата $\mathtt{z}$ и тя е свободна. Щом това срещане на $\mathtt{z}$ е свободно в $\forall \mathtt{x}\, \psi$, то същото срещане ще е свободно и в ψ. Във втория случай началният квантор $\mathtt{x}$ управлява $\mathtt{z}$. Тъй като формулите $\forall \mathtt{x}\, \varphi$ и $\forall \mathtt{x}\, \psi$ са конгруентни, то във втората променлива на позицията на $\mathtt{z}$ също стои

променлива, която се управлява от началния квантор $\forall \mathtt{x}$. Следователно на тази позиция стои променливата $\mathtt{z} = \mathtt{x}$, която в ψ си няма квантор и значи в ψ е свободна.

Ако $\mathtt{z}$ е свързана променлива в φ, то кванторът, управляващ това срещане на $\mathtt{z}$ в φ, ще го управлява и в $\forall \mathtt{x}\, \varphi$. Тъй като формулите $\forall \mathtt{x}\, \varphi$ и $\forall \mathtt{x}\, \psi$ са конгруентни, то на позицията на $\mathtt{z}$ във втората формула стои променлива $\mathtt{t}$, която е свързана, и управляващият ѝ квантор е на същата позиция, като управляващия квантор на $\mathtt{z}$. Тъй като управляващият квантор на $\mathtt{z}$ бе в φ, то управляващият квантор на $\mathtt{t}$ ще бъде в ψ. Същият квантор ще бъде управляващ на $\mathtt{t}$ и в ψ. ∎

***Задача* 22:** Нека φ е формула, която не съдържа променливата $\mathtt{y}$ и ψ се получава от φ като заменим във φ всяко срещане на променливата $\mathtt{x}$ с $\mathtt{y}$. Да се докаже, че всяка свободна променлива на φ, която е различна от $\mathtt{x}$, е свободна променлива и на ψ.

***Задача* 23:** Нека φ съдържа подформула ψ и ψ' е конгруентна на ψ. Нека φ' се получава от φ като заменим подформулата ψ с ψ'. Да се докаже, че φ и φ' са конгруентни.

***Задача* 24:** Да се докаже, че

а) $\neg\varphi \equiv \neg\varphi' \longleftrightarrow \varphi \equiv \varphi'$

б) $\varphi \,\&\, \psi \equiv \varphi' \,\&\, \psi' \longleftrightarrow \varphi \equiv \varphi'$ и $\psi \equiv \psi'$

в) $\varphi \vee \psi \equiv \varphi' \vee \psi' \longleftrightarrow \varphi \equiv \varphi'$ и $\psi \equiv \psi'$

г) $\varphi \Rightarrow \psi \equiv \varphi' \Rightarrow \psi' \longleftrightarrow \varphi \equiv \varphi'$ и $\psi \equiv \psi'$

Сравнете тази задача със задача 30.

2.7. Субституции

Субституция в изрази без свързани променливи

В предния раздел казахме, че свързаните променливи могат да се преименуват като това не променя смисъла на терма или формулата в която извършваме преименуването, а свободните променливи не могат да се преименуват без това да смени смисъла. Също така видяхме, че свободните променливи може се заменят с изрази, които не са променливи, докато замяната на една свързана променлива с израз, който не е променлива, със сигурност води до безсмислица.

В предния раздел дадохме точна дефиниция за това как могат да се преименуват свързаните променливи в една формула. Остава да видим

как можем да заменяме свободните променливи с други термове. Найнапред да дефинираме понятието, което ще бъде играе главната роля в този раздел:

✓ **2.7.1. Дефиниция.** *Субституцията* е функция, която на всяка променлива съпоставя терм.

Забележка: Тъй като субституциите се използват в различни алгоритми, дефиницията им обикновено включва допълнителното изискване, че само за краен брой променливи $s(\mathtt{x}) \neq \mathtt{x}$, а за всички останали $s(\mathtt{x}) = \mathtt{x}$. По този начин, за да представим в компютъра една субституция, е достатъчно да помним само двойките $\langle \mathtt{x}, s(\mathtt{x})\rangle$, при които $\mathtt{x} \neq s(\mathtt{x})$, а те са само краен брой. Това допълнително условие усложнява ненужно разсъжденията и затова тук го изпускаме. Въпреки това може да забележим, че субституциите, които се появяват в разглежданите в този курс алгоритми, всъщност удовлетворяват това допълнително изискване.

Тъй като в термовете и безкванторните формули няма свързани променливи, прилагането на субституция към такива изрази става по възможно най-простия и естествен начин — ако s е субституция, просто трябва да заменим в израза всяка променлива $\mathtt{x}$ с $s(\mathtt{x})$:

✓ **2.7.2. Дефиниция.** Ако s е субституция, а τ — терм или предикатна формула без квантори, то *резултатът от прилагането на субституцията s към τ* се получава като заместим в τ всяко срещане на променливата $\mathtt{x}$ с терма $s(\mathtt{x})$.

2.7.3. Означение. За резултата от прилагане на субституция s към израз τ най-често се използва означението τs. Например ако $\mathtt{x}$ е променлива, то $\mathtt{x}s = s(\mathtt{x})$.

2.7.4. Твърдение. *Нека s е субституция.*

а) Ако τ е терм, то τs е терм.

б) Ако φ е безкванторна формула, то φs е безкванторна формула.

Доказателство. Твърдението може да се докаже директно с помощта на индукция. Освен това то следва и от по-общата лема 2.7.5, която ще използваме и по-нататък. ∎

2.7.5. Лема. *а) Ако τ е терм и изразът τ' се получава като по напълно произволен начин заменим в τ някои променливи с термове, то τ' е терм.*

б) Ако φ е формула и изразът φ' се получава като по напълно произволен начин заменим в φ с термове някои от променливите, непосредствено пред които не стоят символите $\forall$ или $\exists$, то φ' е формула.

Доказателство. **(а)** С индукция по броя на символите в терма τ. Ако $\tau = \mathtt{x}$ е променлива, то τ' няма какво друго да бъде, освен $\mathtt{x}$ или терм с който променливата $\mathtt{x}$ е заменена. Ако $\tau = \mathtt{c}$ е константа, то $\tau' = \mathtt{c}$ и е терм. Ако $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$, то τ' има вида $\mathtt{f}(\tau_1', \tau_2', \ldots, \tau_n')$, където $\tau_1', \tau_2', \ldots, \tau_n'$ се получават от $\tau_1, \tau_2, \ldots, \tau_n$ по начина, описан в условието на лемата (заменяме произволни променливи с произволни термове). Съгласно индукционното предположение $\tau_1', \tau_2', \ldots, \tau_n'$ са термове, следователно по дефиниция 2.4.4 в) τ' също е терм.

(б) Аналогично. С индукция по броя на символите във формулата φ. Ако φ е атомарна формула и има вида $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$, то φ' има вида $\mathtt{p}(\tau_1', \tau_2', \ldots, \tau_n')$, където $\tau_1', \tau_2', \ldots, \tau_n'$ се получават от $\tau_1, \tau_2, \ldots, \tau_n$ по начина, описан в условието на лемата (заменяме произволни променливи с произволни термове). Съгласно вече доказаното $\tau_1', \tau_2', \ldots, \tau_n'$ са термове, следователно по дефиниция 2.4.6 φ' също е атомарна формула.

Ако $\varphi = \bot$, то $\varphi' = \bot$ и значи е формула.

Ако $\varphi = \psi_1 \,\&\, \psi_2$, то φ' има вида $\psi_1' \,\&\, \psi_2'$. Съгласно индукционното предположение ψ_1' и ψ_1' са формули, следователно по дефиниция 2.4.10 в) φ' също е формула.

Случаите когато φ има вида $\psi_1 \vee \psi_2$ или $\psi_1 \Rightarrow \psi_2$ се разглеждат аналогично.

Ако $\varphi = \forall \mathtt{x}\, \psi$, то съгласно условието на лемата не можем да заменяме променливата $\mathtt{x}$, пред която стои символът $\forall$, и значи φ' има вида $\forall \mathtt{x}\, \psi'$. Съгласно индукционното предположение ψ' е формула, следователно по дефиниция 2.4.10 г) φ' също е формула.

Случаят когато $\varphi = \exists \mathtt{x}\, \psi$ се разглежда аналогично. ∎

2.7.6. Означение. С $[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n]$ ще означаваме субституцията s, за която

$$s(\xi) = \begin{cases} \tau_1, & \text{ако } \xi = \mathtt{x}_1 \\ \tau_2, & \text{ако } \xi = \mathtt{x}_2 \\ \ldots & \\ \tau_n, & \text{ако } \xi = \mathtt{x}_n \\ \xi, & \text{ако } \xi \notin \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\} \end{cases}$$

2.7.7. Пример. Нека $s = [\mathtt{x}, \mathtt{y} \coloneqq \mathtt{f}(\mathtt{y}), \mathtt{x}]$ и $\tau = \mathtt{g}(\mathtt{x}, \mathtt{y}, \mathtt{f}(\mathtt{x}))$. Тогава $\tau s = \mathtt{g}(\mathtt{f}(\mathtt{y}), \mathtt{x}, \mathtt{f}(\mathtt{f}(\mathtt{y})))$.

✓ ***Задача* 25:** Намерете

$$\mathtt{f}(\mathtt{x}, \mathtt{y})[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{y}, \mathtt{y}]$$
$$\mathtt{f}(\mathtt{x}, \mathtt{y})[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{g}(\mathtt{x}), \mathtt{x}]$$
$$\mathtt{f}(\mathtt{g}(\mathtt{x}), \mathtt{y})[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{g}(\mathtt{x}), \mathtt{x}]$$
$$\mathtt{f}(\mathtt{x}, \mathtt{g}(\mathtt{y}))[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{g}(\mathtt{x}), \mathtt{x}]$$

Прилагане на субституция в общия случай

> *Повечето формулировки на правилото за субституция, които са били публикувани — дори и от най-способните логици — преди 1940, са явно сбъркани.*
>
> Хаскел Къри *и* Робер Фейс *[8]*

Остава да видим как можем да прилагаме субституция към произволна предикатна формула. При термовете и безкванторните формули беше лесно. Например за произволна безкванторна формула φ ако s е субституция, то формулата φs се получава като заменим в φ всяка променлива $\mathtt{x}$ с $s(\mathtt{x})$. Можехме да използваме такава проста дефиниция, защото в термовете, атомарните формули и безкванторните формули няма свързани променливи.

Да видим защо при наличието на свързани променливи прилагането на субституции трябва да се прави по-внимателно. Това е така, защото при прилагането на субституция към израз със свързани променливи може да се допуснат два вида грешки.

✓ **2.7.8. Първия вид грешка** вече я споменахме — при прилагане на субституция не трябва да заменяме свързаните променливи, защото се получават безсмислици. Например ако в израза

$$\sum_{i=1}^{100} i^2$$

заменим i с x^2, ще получим безсмислицата

$$\sum_{x^2=1}^{100} (x^2)^2$$

и ако в предикатната формула

$$\forall \mathtt{x}\, \mathtt{p}(\mathtt{x})$$

заменим $\mathtt{x}$ с терма $\mathtt{f}(\mathtt{y})$, ще получим безсмислицата

$$\forall(\mathtt{f}(\mathtt{y}))\, \mathtt{p}(\mathtt{f}(\mathtt{y}))$$

Извод: Когато прилагаме субституция, трябва да игнорираме променливите, които са свързани от някой квантор. Например ако към формулата

$$\mathtt{p}(\mathtt{x}) \vee \forall \mathtt{x}\, \mathtt{q}(\mathtt{x})$$

искаме да приложим субституция, заменяща променливата $\mathtt{x}$ с терма $\mathtt{f}(\mathtt{y})$, ще получим формулата

$$\mathtt{p}(\mathtt{f}(\mathtt{y})) \vee \forall \mathtt{x}\, \mathtt{q}(\mathtt{x})$$

✓ **2.7.9. Вторият вид грешка** е по-лесно да остане незабелязан и затова трябва да сме внимателни. Да предположим, че сме дефинирали числовата функция f така:

$$f(y) = \int_0^1 (x^2 + y^3)\, dx$$

Това, че в дефиницията на тази функция сме използвали променливата x не означава, че от тук нататък до края на света нямаме право да използваме променливи x за други цели. Да допуснем, че в даден момент сме дефинирали някаква променлива x. На колко ще бъде равно $f(2x)$? Ако просто навсякъде заменим y с $2x$ и кажем, че

$$f(2x) = \int_0^1 (x^2 + (2x)^3)\, dx$$

това ще бъде грешка! Да забележим, че изразът от дясната страна на равенството по никакъв начин не зависи от стойността на x, защото променливата x е свързана. Верният отговор е следният:

$$f(2x) = \int_0^1 (z^2 + (2x)^3)\, dz$$

Разбира се, вместо z можем да използваме коя да е друга променлива, която не се използва за други цели.

Ето друг пример. Нека

$$a_n = \sum_{i=1}^{n} i^3$$

Ако след време дефинираме променлива i, на колко ще бъде равно a_{2i}? Верният отговор е

$$a_{2i} = \sum_{j=1}^{2i} j^3$$

Тук също, вместо j можем да използваме коя да е друга променлива, която не се използва за други цели.

Извод: Когато прилагаме субституция, трябва да преименуваме кванторните променливи, ако има опасност някой квантор да „свърже“ променлива, която ще се постави от субституцията и трябва да остане свободна. Например ако към формулата

$$\forall \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{y})$$

искаме да приложим субституция, заменяща променливата $\mathtt{x}$ с терма $\mathtt{f}(\mathtt{y})$, ще трябва най-напред да преименуваме свързаната променлива $\mathtt{y}$ например на $\mathtt{z}$

$$\forall \mathtt{z}\, \mathtt{q}(\mathtt{x}, \mathtt{z})$$

и едва след това ще може да заменим $\mathtt{x}$ с $\mathtt{f}(\mathtt{y})$:

$$\forall \mathtt{z}\, \mathtt{q}(\mathtt{f}(\mathtt{y}), \mathtt{z})$$

Обмислете добре тези примери, преди да продължите четенето на този раздел! Опитайте се да решите следващата задача и проверете дали отговорът ви е бил верен.

✓ ***Задача* 26:** Още не сме дали точната дефиниция на φs при произволна субституция s и формула φ. Въпреки това, имайки предвид съображенията, изказани дотук, преценете какви трябва да бъдат

а) $(\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \vee \mathtt{q}(\mathtt{x}))[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{y})]$

б) $(\forall \mathtt{y}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \vee \mathtt{q}(\mathtt{y}))[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{y})]$

И така, анализът на грешките от първи и втори тип (вж. 2.7.9 и 2.7.9) показва, че когато субституцията замества едни променливи с термове, съдържащи други променливи, тези променливи — както

заменените, така и новопоявилите се — не трябва да имат нищо общо с кванторните променливи. Ако една заменена променлива е била в областта на действие на квантор, получаваме грешка от първи тип, а когато новопоявила се променлива попадне в областта на действие на квантор, получаваме грешка от втори тип.

За да избегнем грешките от първи тип, в дефиницията трябва да кажем, че заменяме само онези срещания на дадена променлива, които са свободни — в никакъв случай не трябва да променяме свързаните променливи. За да избегнем грешките от втори тип пък, ще трябва да видим кои ще са новите променливи, които ще се появят след като приложим субституцията, и да се погрижим никой от кванторите да не „свърже“ тези нови променливи.

Променливите, които заменя субституцията, са свободните променливи. Ако $\mathtt{z}$ е свободна променлива, тогава субституцията s ще я замени с терма $s(\mathtt{z})$. Следователно новите променливи, които ще се появят след като приложим субституцията, са всички променливи, които се срещат в термовете $s(\mathtt{z})$, където $\mathtt{z}$ е свободна променлива във формулата. Ще наречем такива променливи „опасни“. Изключение е следният очевидно „безопасен“ случай: когато $s(\mathtt{z}) = \mathtt{z}$, тогава заменяме $\mathtt{z}$ с $\mathtt{z}$, т. е. нищо не заменяме и значи не се случва нищо опасно.

✓ **2.7.10. Дефиниция.** Ще наричаме *опасни променливи* за формулата φ при субституция s всички променливи, които се срещат в термовете $s(\mathtt{z})$, където $\mathtt{z}$ е свободна променлива на φ и $s(\mathtt{z}) \neq \mathtt{z}$.

Да забележим, че опасните променливи винаги са краен брой. В предния раздел видяхме, че свързаните променливи могат да се преименуват без това да промени смисъла на формулата. В частност доказахме лема 2.6.11, според която ако си изберем Θ да бъде множеството от опасните променливи, тогава можем да намерим конгруентна формула, която не съдържа нито един квантор, чиято променлива е от това множество. Именно това ни дава възможност да дефинираме прилагането на субституция към произволна формула. Ако във формулата няма квантори с опасни променливи, то няма проблеми да приложим субституцията към всички свободни променливи. В противен случай най-напред намираме конгруентна формула, която не съдържа квантори с опасни променливи и след това прилагаме субституцията към новата формула. Следващата дефиниция формализира всичко това.

✓ **2.7.11. Дефиниция.** Нека s и φ са произволни субституция и формула, а Θ е крайното множество от всички опасни за φ променливи при субституция s. Ако φ не съдържа квантори с променливи от Θ,

нека $\varphi' = \varphi$, а в противен случай нека φ' е конгруентна с φ формула, която не съдържа квантори с променливи от Θ (такава съществува благодарение на лема 2.6.11).

В такъв случай, с φs ще означим израза, който се получава като заместим във φ' всяка срещане на свободна променлива $\mathtt{z}$ с терма $s(\mathtt{z})$. Изразът φs се нарича *резултат от прилагането на субституцията s към φ*.

2.7.12. Да забележим, че според тази дефиниция заменяме само свободните променливи. Ако някое срещане на променливата $\mathtt{z}$ не е свободно, тогава на това място $\mathtt{z}$ не се заменя с терма $s(\mathtt{z})$. В дефиниция 2.7.2 нямаше нужда да правим специална уговорка, че прилагаме субституцията само към свободни променливи, защото в термовете и безкванторните формули всички променливи са свободни. Това впрочем показва, че току-що дадената дефиниция не противоречи на дефиниция 2.7.2, а само я допълва — да заменим всяка променлива $\mathtt{z}$ в терм или безкванторна формула с $s(\mathtt{z})$ е същото каквото да заменим всяка свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$.

✓ **2.7.13. Пример.** Нека s е субституция, за която

$$\begin{aligned} s(\mathtt{x}_1) &= \mathtt{f}(\mathtt{x}_1 + \mathtt{x}_2, \mathtt{z}) \\ s(\mathtt{x}_2) &= \mathtt{y} + \mathtt{x}_3 \\ s(\mathtt{x}_3) &= \mathtt{y} \\ s(\mathtt{y}) &= \mathtt{f}(\mathtt{y}, \mathtt{y}) \\ s(\mathtt{z}) &= (\mathtt{t} + \mathtt{x}_3) + \mathtt{x}_1 \\ s(\mathtt{t}) &= \mathtt{x}_1 + \mathtt{f}(\mathtt{x}_2, \mathtt{t}) \end{aligned}$$

Трябва да приложим тази субституция към формулата

$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{y}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{t}))$$

Тъй като тази формула съдържа квантори, трябва да проверим дали някой от кванторите не е с опасна променлива. Свободни променливи са: $\mathtt{y}, \mathtt{x}_1, \mathtt{t}$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{y}), s(\mathtt{x}_1), s(\mathtt{t})$, т.е. променливите $\mathtt{y}, \mathtt{x}_1, \mathtt{x}_2, \mathtt{z}, \mathtt{t}$. Кванторът $\exists \mathtt{y}$ е с опасна променлива и трябва да го преименуваме. Например ако преименуваме неговата променлива $\mathtt{y}$ на $\mathtt{y}_{178}$ ще получим

$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{y}) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{t}))$$

Към тази формула вече може да прилагаме субституцията s (заменяме само свободните променливи). Получаваме

$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{f}(\mathtt{y}, \mathtt{y})) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_3) = \mathtt{f}(\mathtt{x}_1 + \mathtt{x}_2, \mathtt{z}) + (\mathtt{x}_1 + \mathtt{f}(\mathtt{x}_2, \mathtt{t}))))$$

✓ ***Задача* 27:** Нека s е същата субституция като в пример 2.7.13. Намерете резултатите от прилагането на субституцията s към следните формули:

$$\mathtt{p}(\mathtt{x}_1, \mathtt{z}) \vee \mathtt{x}_1 + \mathtt{x}_2 = \mathtt{f}(\mathtt{t}, \mathtt{y})$$
$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{t}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{t}))$$
$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{z}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{z}))$$
$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{z}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_2 + \mathtt{z}))$$

Свойства на субституциите

> *Това едва ли може да се нарече интуитивна дефиниция за субституция. Тя обаче има точните свойства.*
>
> Джон Рейнолдс *[15]*

В дефиниция 2.7.11 нарекохме φs просто „израз“, а не формула. Всъщност φs е формула, но това се нуждае от доказателство.

2.7.14. Твърдение. *За всяка субституция s и формула φ изразът φs е формула.*

Доказателство. Ако във формулата φ няма квантори с опасни променливи при субституция s, тогава φs се получава просто като заменяме всяко срещане на свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$. Затова искането следва от лема 2.7.5.

Ако във формулата има квантори с опасни променливи, то съгласно дефиниция 2.7.11 трябва най-напред да преименуваме свързаните променливи по подходящ начин, получавайки конгруентна формула φ', след което действаме по вече описания начин. Затова в този случай φs също е формула. ■

Вече споменахме, че ще искаме да отъждествяваме конгруентните формули. Това би създало проблем, ако се окаже, че като приложим някоя субституция към две конгруентни формули, е възможно да получим две съществено различни формули. Следващото твърдение показва, че това за щастие не се случва.

✓ **2.7.15. Твърдение.** *Ако $\varphi \equiv \psi$, то $\varphi s \equiv \psi s$. С други думи, ако формулите φ и ψ са конгруентни, то за произволна субституция s, формулите φs и ψs са конгруентни.*

Доказателство. Ще докажем твърдението най-напред за частния случай когато φ и ψ не съдържат квантори с опасни променливи при субституция s. В този случай φs и ψs се получават като заменим в тях всяка свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$. Тъй като φ и ψ са конгруентни, то свободните променливи в тях са едни и същи и се намират на една и съща позиция. След като забелязахме тези предварителни неща, да видим, че условията на дефиницията за конгруентност са изпълнени в случая на φs и ψs.

Двете формули са с една и съща дължина, защото φ и ψ са с една и съща дължина, след което в тях едни и същи променливи заменяме с едни и същи термове. По същата причина в φs и ψs символите, които не са променливи, съвпадат.

Нека $\mathtt{x}$ е свободна променлива в φs. Тогава това срещане на $\mathtt{x}$ се намира в някой от термовете $s(\mathtt{z})$, с които сме заменили свободната променлива $\mathtt{z}$ в φ. Но същата променлива $\mathtt{z}$ се намира на същата позиция и в ψ, и то отново като свободна и значи там също я заменяме с терма $s(\mathtt{z})$. Следователно на позицията, където в φs се среща $\mathtt{x}$, в ψs на същата позиция също се среща $\mathtt{x}$. Това срещане на $\mathtt{x}$ е свободно, защото в противен случай в ψs би имало квантор, който управлява $\mathtt{x}$ и значи би имало квантор с опасна променлива, а такъв квантор се уговорихме, че няма.

Нека $\mathtt{x}$ е свързана променлива в φs. Това срещане на $\mathtt{x}$ не може да се намира в термовете $s(\mathtt{z})$, с които заменяме свободните променливи $\mathtt{z}$ в φ, защото в противен случай в φs би имало квантор, който управлява $\mathtt{x}$ и значи би имало квантор с опасна променлива. Следователно променливата $\mathtt{x}$ се среща извън термовете $s(\mathtt{z})$, с които заменяме свободните променливи в φ, което означава, че $\mathtt{x}$ е свързана и в φ, защото само свързаните променливи в φ не са заменени с такива термове. Щом тази променлива е свързана в φ, то значи си има управляващ квантор. От конгруентността на φ и ψ следва че на позицията, на която се среща $\mathtt{x}$ в φ, във формулата ψ също стои променлива, да я наречем $\mathtt{y}$, и управляващият квантор на $\mathtt{y}$ се намира в ψ на същата позиция, на която е и управляващият квантор на $\mathtt{x}$ в φ. Щом $\mathtt{y}$ е свързана в ψ, то значи субституцията s не я променя и значи $\mathtt{y}$ остава заедно с управляващия си квантор и в ψs.

Доказахме частния случай на твърдението. Общият следва от него. Съгласно дефиницията за прилагане на субституция към формула (2.7.11) съществуват такива конгруентна на φ формула φ' и конгруентна на ψ формула ψ', че $\varphi s = \varphi' s$, $\psi s = \psi' s$ и във формулите φ' и ψ' няма квантори с опасни променливи. Значи от вече доказания частен случай получаваме, че $\varphi' s$ и $\psi' s$ са конгруентни. ∎

✓ **2.7.16. Твърдение.** *Нека субституциите s_1 и s_2 съвпадат за всички свободни променливи на формулата φ. Тогава $\varphi s_1 \equiv \varphi s_2$.*

Доказателство. Нека φ' е конгруентна с φ формула, която не съдържа квантори с променливи, които са опасни при субституция s_1. Формулата φ' има същите свободни променливи като φ, а за всяка такава свободна променлива субституциите s_1 и s_2 съвпадат. Това означава, че формулата φ' не съдържа квантори с опасни променливи също и при субституция s_2. От дефиницията за прилагане на субституция (дефиниция 2.7.11) следва, че $\varphi' s_1 = \varphi' s_2$.

От тук получаваме исканото, защото от една страна $\varphi s_1 \equiv \varphi' s_1$, а от друга $\varphi' s_2 \equiv \varphi s_2$. ∎

2.7.17. Твърдение. *Ако са дадени две субституции s_1 и s_2, да дефинираме s да бъде субституцията, за която $s(\mathtt{z}) = (\mathtt{z}s_1)s_2$ за всяка променлива $\mathtt{z}$. Тогава за всяка формула φ*

$$\varphi s \equiv (\varphi s_1)s_2$$

Доказателство. Нека s е субституцията, за която $s(\mathtt{z}) = (\mathtt{z}s_1)s_2$ за всяка променлива $\mathtt{z}$. Ще докажем, че за произволна формула $\varphi s \equiv (\varphi s_1)s_2$.

Нека Θ е множеството от свободните променливи на φ. Нека Θ_1 е обединението на Θ с множеството от всички променливи, които се срещат в термовете $s_1(\mathtt{z})$, където $\mathtt{z} \in \Theta$. В такъв случай Θ_1 ще включва всички опасни променливи в φ при субституция s_1. Нека Θ_2 е обединението на Θ_1 с множеството от всички променливи, които се срещат в термовете $s_2(\mathtt{z})$, където $\mathtt{z} \in \Theta_1$. Множеството Θ_2 е крайно, значи от лемата за преименуване на свързаните променливи 2.6.11 можем да намерим формула ψ, която е конгруентна с φ, и която не съдържа квантори с променливи от Θ_2.

Тъй като Θ_1 съдържа опасните променливи за φ при субституция s_1 и $\Theta_1 \subseteq \Theta_2$, то ψ не съдържа квантори с променливи, които са опасни при субституция s_1. Следователно ψs_1 се получава като заменим в ψ всяка свободна променлива $\mathtt{x}$ с терма $s_1(\mathtt{x})$. Това означава, че свободните променливи на ψs_1 са елементи на Θ_1, а формулата ψs_1 не съдържа квантори с променливи, които са опасни при субституция s_2, следователно $(\psi s_1)s_2$ се получава от ψs_1 като заменим всяка свободна променлива $\mathtt{y}$ с терма $s_2(\mathtt{y})$. Тъй като всички свободни променливи в ψs_1 се съдържат в термовете $s_1(\mathtt{x})$, които са заменили свободните променливи $\mathtt{x}$ в ψ, то това означава, че $(\psi s_1)s_2$ може да се получи от ψ като заменим всяка свободна променлива $\mathtt{x}$ в ψ с терма $(\mathtt{x}s_1)s_1$.

Тъй като ψ не съдържа квантори с променливи от Θ_2, то ψ не съдържа квантори с променливи, които са опасни при субституция s. Следователно формулата ψs може да се получи от ψ като заменим всяка свободна променлива $\mathtt{x}$ в ψ с терма $s(\mathtt{x})$, т.е. с терма $(\mathtt{x}s_1)s_1$.

Доказахме, че $\psi s = (\psi s_1)s_2$, откъдето следва $\varphi s \equiv (\varphi s_1)s_2$. ∎

***Задача* 28:** Да се докаже, че $\mathtt{x}$ е свободна променлива на формулата φs тогава и само тогава, когато $\mathtt{x}$ се среща в терм от вида $s(\mathtt{z})$, където $\mathtt{z}$ е свободна променлива на φ.

***Задача* 29:** Да се докаже, че

а) $(\neg\varphi)s \equiv \neg(\varphi s)$

б) $(\varphi \,\&\, \psi)s \equiv \varphi s \,\&\, \psi s$

в) $(\varphi \vee \psi)s \equiv \varphi s \vee \psi s$

г) $(\varphi \Rightarrow \psi)s \equiv \varphi s \Rightarrow \psi s$

****Задача* 30:** Да се докаже, че $\forall\mathtt{x}\,\varphi \equiv \forall\mathtt{y}\,\psi$ тогава и само тогава, когато $\mathtt{y}$ не е свободна променлива на $\forall\mathtt{x}\,\varphi$ и $\psi \equiv \varphi[\mathtt{x} \coloneqq \mathtt{y}]$. Сравнете тази задача със задача 24.

****Задача* 31:** Нека променливата $\mathtt{y}$ не се среща никъде във формулата φ и формулата ψ се получава от φ като заменим всяко срещане (свободно и свързано) на променливата $\mathtt{x}$ с $\mathtt{y}$. Да се докаже, че $\psi \equiv \varphi[\mathtt{x} \coloneqq \mathtt{y}]$.

****Задача* 32:** Да се докаже, че ако $\forall\mathtt{x}\,\varphi \equiv \forall\mathtt{x}'\varphi'$ и $\forall\mathtt{x}\,\psi \equiv \forall\mathtt{x}'\psi'$, то $\forall\mathtt{x}\,(\varphi \,\&\, \psi) \equiv \forall\mathtt{x}'(\varphi' \,\&\, \psi')$.

Глава 3

Семантика

Семантиките са странен вид приложна математика; тя се интересува от прозорливи дефиниции вместо от трудни теореми.

Джон Рейнолдс *(според [19, стр. 3])*

3.1. Структури

Вече имахме възможност да работим неформално с някои структури. Например всички разгледани моноиди и полупръстени бяха примери за структури. Видяхме, че един и същи терм може да има различни стойности в различните структури, а една формула може да бъде вярна в една структура и невярна в друга. Все още обаче не сме дали точна математическа дефиниция на това що е структура. В този раздел ще направим това.

Тъй като моноидите представляват прости примери за структури, нека ги използваме, за да се подсетим как трябва да изглежда точната дефиниция на „структура“. Алгебристите дефинират моноида като множество, снабдено с асоциативна операция с неутрален елемент. Асоциативната операция често се означава със символа за умножение, а неутралният елемент — със символа 1. Ето два прости примера за моноиди:

– Положителните реални числа като моноидната операция е умножението, а неутрален елемент е числото едно.
– Естествените числа като моноидната операция е събирането, а неутрален елемент е числото нула.

Тези два примера ни подсказват, че структурата трябва да определя следните неща:

– Множество от елементи, което ще наричаме *универсум*. Например в първия от горните два примера универсум са положителните реални числа, а във втория пример — естествените числа.
– Интерпретация на символите за константа. Например символът „1“ в първия моноид се интерпретира като числото едно, а във втория — като числото нула.
– Интепретация на функционалните символи. Например символът „.“ в първия моноид се интерпретира като умножение, а във втория — като събиране.

Ако сега дадем дефиницията на структура въз основа на така направените наблюдения, ще получим това, което обикновено се нарича *алгебра* или *алгебрична структура* и представлява специален случай (макар и важен) на по-общото понятие за структура. Освен споменатите по-горе неща, структурите трябва да определят още и интерпретацията на предикатните символи. Причината, поради която моноидите не можаха да ни подскажат, че структурите трябва да определят и интерпретацията на предикатните символи, е това, че при тях единственият предикатен символ е $=$, а за този предикатен символ обикновено се очаква да бъде интерпретиран като равенство, а не по някакъв друг странен начин.

В предната глава казахме, че за да си спестим някои усложнения, тук ще използваме с едносортната предикатна логика, в която се използват едносортни структури. При многосортната предикатна логика се използват многосортни структури, които определят какви са елементите на всеки от сортовете. Например линейните пространства представляват двусортни структури, чийто универсум се състои не от едно, а от две множества — елементите от сорт **вектор** и елементите от сорт **скалар**. Тези две множества се наричат *носители*. Тъй като нашите структури са едносортни, то универсумът им се състои от един носител. Поради това при едносортните структури, които тук ще използваме, понятията „универсум“ и „носител“ са взаимнозаменяеми.

Забележка: Въпреки че нашите структури също имат два сорта — **индив** и **съжд** — структурите ще игнорират сорта **съжд** и няма да

определят неговия носител. Това е така, защото този сорт ще има един и същи смисъл във всички структури.*

3.1.1. Означение.

а) Множеството от всички неща от сорт **индив** в структура $\mathbf{M}$ се означава с $[\![\textbf{индив}]\!]^{\mathbf{M}}$ или с $|\mathbf{M}|$ и се нарича *универсум* или *носител* на структурата.**

б) *Смисълът* или *интерпретацията* в дадена структура $\mathbf{M}$ на символите за константи, функционалните символи и предикатните символи ще означаваме, използвайки семантичните скоби $[\![\cdot]\!]^{\mathbf{M}}$ или посредством горен индекс $^{\mathbf{M}}$. Например:

- $[\![\mathtt{c}]\!]^{\mathbf{M}}$ или $\mathtt{c}^{\mathbf{M}}$ е интерпретацията на символа за константа $\mathtt{c}$ в структурата $\mathbf{M}$,
- $[\![\mathtt{f}]\!]^{\mathbf{M}}$ или $\mathtt{f}^{\mathbf{M}}$ е интерпретацията на функционалния символ $\mathtt{f}$ в структурата $\mathbf{M}$ и
- $[\![\mathtt{p}]\!]^{\mathbf{M}}$ или $\mathtt{p}^{\mathbf{M}}$ е интерпретацията на предикатния символ $\mathtt{p}$ в структурата $\mathbf{M}$.

По отношение на това какъв точно може да бъде универсумът на една структура и как трябва да се интерпретират различните символи в нея, да обърнем внимание на следното.

Универсумът е множество. Не се налага да ограничаваме по какъвто и да е начин какво точно съдържа това множесто — елементите му могат да бъдат числа, функции, други множества и т. н. Единственото ограничение, което се налага да приемем, е това универсумът да бъде непразно множество. Причината, поради която налагаме това ограничение, е следната — когато универсумът е празното множество, се появяват някои странности, които неужно ще усложняват формулировките на твърденията и техните доказателства. И тъй като структурите с празен универсум очевидно не могат да бъдат кой знае колко полезни, струва си да си спестим тези усложнения като забраним структурите с празен универсум.

*Понякога обаче се оказват полезни и структури с нестандартен носител за сорта **съжд**.

**Някои математици използват за структурите удебелени латински букви — $\mathbf{A}$, $\mathbf{B}$, $\mathbf{M}$, $\mathbf{N}$, други калиграфски букви — $\mathcal{A}$, $\mathcal{B}$, $\mathcal{M}$, $\mathcal{N}$, трети готически — $\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{M}$, $\mathfrak{N}$. Универсумът на структурите се означава или с вертикални черти — $|\mathbf{A}|$, $|\mathbf{B}|$, $|\mathbf{M}|$, $|\mathbf{N}|$, или посредством съответните обикновени латински букви — A, B, M, N.

Смисълът или интерпретацията на символите за константи трябва да бъде елемент на универсума на структурата. Например ако универсумът $|\mathbf{M}|$ на структурата $\mathbf{M}$ е множеството на реалните числа и c е символ за константа, тогава $[\![\mathsf{c}]\!]^{\mathbf{M}}$ трябва да бъде реално число. Ако пък универсумът е множеството на естествените числа, тогава $[\![\mathsf{c}]\!]^{\mathbf{M}}$ трябва да бъде естествено число. И т. н.

Тъй като по дефиниция типът на n-местните функционални символи е

$$\langle \underbrace{\textbf{индив}, \textbf{индив}, \ldots, \textbf{индив}}_{n \text{ пъти } \textbf{индив}} \rangle \mapsto \textbf{индив}$$

то е естествено такъв символ да бъде интерпретиран в структурата като функция с n аргумента. Както аргументите, така и стойността на тази функция са елементи на универсума $[\![\textbf{индив}]\!]^{\mathbf{M}}$. Например в първия от по-горните два моноида двуместният функционален символ „.“ се интерпретираше функцията умножение на две реални числа, а във втория — като функцията събиране на естествени числа.

Тъй като типът на един n-местен предикатен символ е

$$\langle \underbrace{\textbf{индив}, \textbf{индив}, \ldots, \textbf{индив}}_{n \text{ броя } \textbf{индив}} \rangle \mapsto \textbf{съжд}$$

то е естествено и такъв символ да бъде интерпретиран в структурата като функция с n аргумента. Също като при функционалните символи, аргументите на тази функция трябва да бъдат елементи на универсума $[\![\textbf{индив}]\!]^{\mathbf{M}}$. Стойността на тази функция обаче трябва да бъде не елемент на универсума, а съждение.

Да разгледаме по-конкретен пример. Казахме, че при алгебричните структури има двуместен предикатен символ $=$, който се интерпретира като равенство. Това означава, че ако някоя алгебрична структура $\mathbf{A}$ има за универсум реалните числа, тогава символът $=$ в нея се интерпратира като двуаргументна функция $[\![=]\!]^{\mathbf{A}}$, чиито аргументи са реални числа, а връщаната стойност — съждение. Например ако дадем на функцията $[\![=]\!]^{\mathbf{A}}$ като аргументи числата 2 и 5, тя ще ни върне като стойност съждението „числото 2 е равно на 5“, ако пък ѝ дадем като артументи числата -3 и -3, ще върне съждението „числото -3 е равно на -3“ и т.н.

Нека структурата $\mathbf{M}$ има за универсум реалните числа и сигнатурата ѝ включва двуместния предикатен символ $<$. Ако кажем, че в тази структура символът $<$ се интерпретира като „по-малко“, това всъщност означава, че на символа $<$ в структурата съответства двуаргументната функция $[\![<]\!]^{\mathbf{M}}$, на която ако ѝ дадем като аргументи например 2 и 8,

ще ни върне като стойност съждението „2 е по-малко от 8“. Но няма никакъв проблем в друга структура $\mathbf{N}$ символът $<$ да се интерпретира не като „по-малко“, а като „по-голямо“. Тогава съответната функция $[\![<]\!]^{\mathbf{N}}$ ще ни върне като стойност съждението „2 е по-голямо от 8“.

В съвременната математическа традиция не е прието съжденията да се използват като пълноправни математически обекти. Това означава, че дадената по-горе дефиниция за интерпретацията на n-местните предикатни символи е донякъде неприемлива. Има два начина да избегнем това затруднение. Първият начин е да направим функцията да връща като стойности не съждения, а „по-приемливи“ обекти като „И“ или „Л“, „`true`“ или „`false`“, 1 или 0. Вторият начин е да забележим, че вместо съждения може да използваме множества. В нашия случай може да интерпретираме един n-местен предикатен символ посредством множеството от всички n-торки, за които предикатният символ трябва да бъде верен в структурата.

3.1.2. Дефиниция. Нека X е произволно множество.

а) n-местна *функция* в X означава n-местна функция с аргументи от X и стойност в X.

б) n-местен *предикат* в X означава n-местна функция с аргументи от X, която приема като стойност някое съждение.

в) n-местна *релация* в X е подмножество на $X^n = \underbrace{X \times X \times \cdots \times X}_{n \text{ пъти}}$.

Ако решим да интерпретираме предикатните символи не с предикати, а с релации, то например предикатният символ $=$ ще се интерпретира не посредством функцията, която по аргументи x и y връща като стойност съждението „x е равно на y“, а посредством множеството от всички двойки $\langle x, y\rangle$, за които е вярно това съждение.

3.1.3. Двата начина за дефиниране на семантиката на предикатните символи — със съждения и с множества — са взаимнозаменяеми. Например вместо да използваме двуаргументна функция, която по дадени като аргументи две числа x и y връща съждението „$x < y$“, може да използваме множеството от всички двойки $\langle x, y\rangle$, за които това съждение е вярно. И обратно, ако ни е дадено множество P от двойки $\langle x, y\rangle$, тогава може да дефинираме функция, която по дадени аргументи x и y връща като стойност съждението „$\langle x, y\rangle \in P$“.

За да избегнем използването на съждения като пълноправни математически обекти, в този курс ще интерпретираме предикатните символи посредством релации. Следващото означение обаче ще ни даде

възможност да забравим, че интерпретираме предикатните символи посредством релации, и да работим с тях така, все едно че ги интерпретираме посредством предикати:

3.1.4. Означение. Ако R е множество от n-торки, то ще използваме изрази от вида

$$R(x_1, x_2, \ldots, x_n)$$

като съкратен запис на

$$\langle x_1, x_2, \ldots, x_n \rangle \in R$$

Благодарение на това означение за всеки предикатен символ $\mathtt{p}$ може да използваме записа $\mathtt{p}^{\mathbf{M}}(x_1, x_2, \ldots, x_n)$, т. е. да го използваме така, все едно че $\mathtt{p}^{\mathbf{M}}$ е функция. В действителност $\mathtt{p}^{\mathbf{M}}(x_1, x_2, \ldots, x_n)$ е съкратен запис на $\langle x_1, x_2, \ldots, x_n \rangle \in \mathtt{p}^{\mathbf{M}}$, но обикновено не се налага да помним това.

Вече сме готови да дадем точната дефиниция на структура за сигнатурата $\mathfrak{sig} = \langle \mathtt{d}_1, \mathtt{d}_2, \mathtt{d}_3, \ldots \rangle$. Това ще бъде редица, в която първият елемент е универсумът на структурата (т. е. множеството от индивидите от сорт **индив**), а останалите елементи са интерпретациите на символите $\mathtt{d}_1, \mathtt{d}_2, \mathtt{d}_3, \ldots$.

✓ **3.1.5. Дефиниция.** Нека $\mathfrak{sig} = \langle \mathtt{d}_1, \mathtt{d}_2, \mathtt{d}_3, \ldots \rangle$ е сигнатура. Редицата $\mathbf{M} = \langle [\![\textbf{индив}]\!]^{\mathbf{M}}, [\![\mathtt{d}_1]\!]^{\mathbf{M}}, [\![\mathtt{d}_2]\!]^{\mathbf{M}}, [\![\mathtt{d}_3]\!]^{\mathbf{M}}, \ldots \rangle$ е *структура* за $\mathfrak{sig}$, ако $[\![\textbf{индив}]\!]^{\mathbf{M}}$ е непразно множество, наречено *универсум* или *носител* на структурата. Освен това:

а) Ако $\mathtt{d}_i$ е символ за константа, то интерпретацията $[\![\mathtt{d}_i]\!]^{\mathbf{M}}$ на $\mathtt{d}_i$ е елемент на универсума.

б) Ако $\mathtt{d}$ е n-местен функционален символ, то интерпретацията $[\![\mathtt{d}_i]\!]^{\mathbf{M}}$ на $\mathtt{d}_i$ е n-местна функция в универсума.

в) Ако $\mathtt{d}$ е n-местен предикатен символ, то интерпретацията $[\![\mathtt{d}_i]\!]^{\mathbf{M}}$ на $\mathtt{d}_i$ е n-местна релация в универсума.

Универсумът на структурата $\mathbf{M}$ се означава с $|\mathbf{M}|$ или $[\![\textbf{индив}]\!]^{\mathbf{M}}$. Интерпретацията на символа $\mathtt{d}$ се означава с $\mathtt{d}^{\mathbf{M}}$ или $[\![\mathtt{d}]\!]^{\mathbf{M}}$.

Интерпретация на символите за константи и функционалните символи в структура:

$$\mathtt{c}^{\mathbf{M}} \in |\mathbf{M}|$$
$$\mathtt{f}^{\mathbf{M}} \colon |\mathbf{M}|^n \to |\mathbf{M}|$$

Интерпретация на предикатните символи:

$$\mathtt{p}^{\mathbf{M}} \colon |\mathbf{M}|^n \to \text{ съждение} \qquad \text{(с предикат)}$$
$$\mathtt{p}^{\mathbf{M}} \subseteq |\mathbf{M}|^n \qquad \text{(с релация)}$$

3.1.6. Дефиниция. а) Когато сигнатурата съдържа двуместен предикатен символ $=$, а структурата интерпретира този символ като равенство, то за тази структура казваме, че е *структура с равенство*.

б) Когато сигнатурата е алгебрична (т.е. единственият предикатен символ в нея е $=$) и структурата интерпретира този символ като равенство, тогава такава структура се нарича *алгебрична структура* или просто *алгебра*. Причината за тази терминология е това, че повечето от структурите, които се използват в алгебрата, са точно такива.*

3.1.7. Пример. Магмите, полугрупите, моноидите, групите, квазигрупите, полупръстените, пръстените, пръстените на Ли, полетата, алгебрите на Клини, полурешетките, решетките и булевите алгебри са само някои от многото примери за алгебрични структури. Например пръстенът на целите числа е структура, чийто универсум е множеството на целите числа, има два символа за константи 0 и 1, които се интерпретират стандартно като числото нула и числото едно, един едноместен функционален символ „$-$“, който се интерпретира като операцията „смяна на знака“, два двуместни функционални символа „$+$“ и „$.$“, които се интерпретират като операциите събиране и умножение и един предикатен символ $=$, който е двуместен и се интерпретира като равенство.

3.1.8. Пример. Графите са пример за структури, които не са алгебрични. Един *граф* $\mathbf{G}$ може да се мисли като структура, чийто универсум е множеството от върховете на графа, символи за константи и функ-

*Понякога структурите в алгебрата са снабдени с някаква наредба. Строго погледнато, такива структури не са алгебрични.

ционални символи няма и има единствен предикатен символ е $\mathtt{p}$, който е двуместен и за всеки два върха v_1 и v_2:

- ако си мислим, че $\mathtt{p}$ се интерпретира с предикат, то $\mathtt{p}^{\mathbf{G}}(v_1, v_2)$ е истина тогава и само тогава, когато има ребро от v_1 до v_2;
- ако си мислим, че $\mathtt{p}$ се интерпретира с релация, то $\langle v_1, v_2 \rangle \in \mathtt{p}^{\mathbf{G}}$ е истина тогава и само тогава, когато има ребро от v_1 до v_2, т. е.

$$\mathtt{p}^{\mathbf{G}} = \{\, \langle v_1, v_2 \rangle \mid \text{има ребро от } v_1 \text{ до } v_2 \,\}$$

3.1.9. Пример. Нека сигнатурата $\mathfrak{sig}$ е такава, че в нея има единствен символ за константа $\mathtt{c}$, единствен функционален символ $\mathtt{f}$, който е двуместен, и единствен предикатен символ $\mathtt{p}$, който също е двуместен. Нека структурата $\mathbf{M}$ за $\mathfrak{sig}$ е с универсум множеството на реалните числа, $\mathtt{c}^{\mathbf{M}} = 3$, $\mathtt{f}^{\mathbf{M}}(x, y) = x + y$ и $\mathtt{p}^{\mathbf{M}}(x, y)$ е съждението „$x < y$“. В такъв случай формулата

$$\mathtt{p}(\mathtt{c}, \mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{c}, \mathtt{f}(\mathtt{x}, \mathtt{x}))$$

казва, че ако едно реално число x е по-голямо от 3, то $x + x$ също е по-голямо от 3.

✓ ***Задача* 33:** Какво казват в структурата от пример 3.1.9 следните формули:

$$\mathtt{p}(\mathtt{x}, \mathtt{y}) \,\&\, \mathtt{p}(\mathtt{y}, \mathtt{z}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{z}) \tag{6}$$

$$\mathtt{p}(\mathtt{x}, \mathtt{y}) \Rightarrow \mathtt{p}(\mathtt{f}(\mathtt{x}, \mathtt{z}), \mathtt{f}(\mathtt{y}, \mathtt{z})) \tag{7}$$

$$\mathtt{p}(\mathtt{c}, \mathtt{c}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{x}) \tag{8}$$

***Задача* 34:** В пример 3.1.8 видяхме, че графите може да се мислят като специален вид структури. Напишете формула, която е вярна в един граф тогава и само тогава, когато

а) графът е неориентиран;

б) графът е пълен

3.2. Оценки

В предния раздел дадохме дефиницията на структура, но все още не сме дали точна дефиниция за това какво значи една формула да бъде вярна в структура. В пример 3.1.9 разчитахме повече на интуицията си и здравия смисъл, отколкото на някаква точна математическа дефиниция.

За да дефинираме какво значи една формула да бъде вярна структура, е нужно да се научим да намираме стойността на термовете в структурата. За да пресметнем стойността на един терм обаче, не е достатъчно да знаем коя е структурата. Да разгледаме например терма $\mathtt{x} + \mathtt{x}$. Структурата ще ни каже какъв е смисълът на функционалният символ $+$, но не и стойността на променливата $\mathtt{x}$. А при различни стойности на променливата $\mathtt{x}$, стойността на този терм най-вероятно ще бъде различна. Затова ще дефинираме понятието „оценка на променливите“:

✓ **3.2.1. Дефиниция.** Функцията v е *оценка* в структурата $\mathbf{M}$, ако v съпоставя на всяка променлива елемент на универсума на $\mathbf{M}$.

Нека например структурата $\mathbf{M}$ е с универсум $\mathbb{R}$ и $+^{\mathbf{M}}$ е функцията събиране на реални числа. Нека оценката v в $\mathbf{M}$ е такава, че $v(\mathtt{x}) = 5$ и $v(\mathtt{y}) = 3$. Тогава стойността на терма $\mathtt{x} + \mathtt{y}$ в структурата $\mathbf{M}$ при оценка v трябва да бъде числото 8. Да забележим, че за да кажем каква трябва да бъде стойността на терма $\mathtt{x} + \mathtt{y}$, не бе нужно да знаем на колко е равно $v(\mathtt{z})$ за променливи $\mathtt{z}$, различни от $\mathtt{x}$ и $\mathtt{y}$. Единствените променливи, които имат отношение към стойността на терма $\mathtt{x} + \mathtt{y}$ са променливите, срещащи се в този терм, т.е. $\mathtt{x}$ и $\mathtt{y}$.

***Задача* 35:** Докажете, че за произволна структура $\mathbf{M}$ съществува поне една оценка в $\mathbf{M}$.

Точната дефиниция за стойност на терм е следната:

✓ **3.2.2. Дефиниция.** Нека $\mathbf{M}$ е структура за сигнатурата $\mathfrak{sig}$ и v е оценка в $\mathbf{M}$. *Стойността* в структурата $\mathbf{M}$ при оценка v на термовете при сигнатура $\mathfrak{sig}$ се дефинира индуктивно:

а) ако $\mathtt{x}$ е променлива, то стойността на терма $\mathtt{x}$ е $v(\mathtt{x})$;

б) ако $\mathtt{c}$ е символ за константа, то стойността на терма $\mathtt{c}$ е $\mathtt{c}^{\mathbf{M}}$;

в) ако $\mathtt{f}$ е n-местен функционален символ и $\tau_1, \tau_2, \ldots, \tau_n$ са термове, то стойността на терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ е равна на $\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)$, където μ_i е стойността на τ_i в $\mathbf{M}$ при оценка v.

Да обърнем внимание, че стойността на терм в структура е дефинирана само ако термът и структурата са за една и съща сигнатура. Например няма как да пресметнем стойността на терма $\mathtt{x}+\mathtt{y}$, ако структурата е за сигнатура, в която няма функционален символ $+$. Стойността на атомарна формула също е дефинирана само ако атомарната формула и структурата са за една и съща сигнатура.

3.2.3. Означение. Нека $\mathbf{M}$ е структура и τ е терм. *Смисълът* или *стойността* на терма τ в структурата $\mathbf{M}$ е функция, означавана с $[\![\tau]\!]^{\mathbf{M}}$, чийто аргумент е оценка v в $\mathbf{M}$, а стойността ѝ е равна на стойността на τ в структурата $\mathbf{M}$ при оценка v. Следователно можем да означаваме стойността на терм τ в структура $\mathbf{M}$ при оценка v посредством семантичните скоби $[\![\tau]\!]^{\mathbf{M}}v$.*

3.2.4. *Забележка*: Използвайки току-що даденото означение, можем да формулираме точките от дефиницията за стойност на терм по следния начин:

- стойността $[\![\mathtt{x}]\!]^{\mathbf{M}}v$ на променлива $\mathtt{x}$ се определя от оценката v и е равна на $v(\mathtt{x})$;
- стойността $[\![\mathtt{c}]\!]^{\mathbf{M}}$ на символ за константа $\mathtt{c}$ се определя от структурата;
- стойността на терм от вида $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ се получава като „спускаме“ рекурсивно семантичните скоби $[\![\cdot]\!]^{\mathbf{M}}$:

$$[\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)]\!]^{\mathbf{M}}v = [\![\mathtt{f}]\!]^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v, [\![\tau_2]\!]^{\mathbf{M}}v, \ldots, [\![\tau_n]\!]^{\mathbf{M}}v)$$

Например

$$[\![\mathtt{g}(\mathtt{f}(\mathtt{x}, \mathtt{c}), \mathtt{y})]\!]^{\mathbf{M}}v = [\![\mathtt{g}]\!]^{\mathbf{M}}([\![\mathtt{f}]\!]^{\mathbf{M}}(v(\mathtt{x}), [\![\mathtt{c}]\!]^{\mathbf{M}}), v(\mathtt{y})) = \mathtt{g}^{\mathbf{M}}(\mathtt{f}^{\mathbf{M}}(v(\mathtt{x}), \mathtt{c}^{\mathbf{M}}), v(\mathtt{y}))$$

Едно важно свойство на стойността на терм е това, че стойността не зависи от променливите, които не се срещат в терма. Например за да пресметнем стойността на терма $\mathtt{x} + (\mathtt{y} + \mathtt{y})$, не е нужно да знаем каква стойност има променливата $\mathtt{z}$. Да докажем този факт.

✓ **3.2.5. Твърдение.** *Нека v и v' са оценки в структурата $\mathbf{M}$. Ако $v(\mathtt{x}) = v'(\mathtt{x})$ за всяка променлива $\mathtt{x}$, която се среща в терма τ, то $[\![\tau]\!]^{\mathbf{M}}v = [\![\tau]\!]^{\mathbf{M}}v'$.*

✓ Доказателство. С индукция по терма τ, използвайки изискванията на дефиницията за стойност на терм (вж. забележка 3.2.4).

Ако $\tau = \mathtt{x}$ е променлива, то $[\![\tau]\!]^{\mathbf{M}}v = [\![\mathtt{x}]\!]^{\mathbf{M}}v = v(\mathtt{x})$. Тъй като по условие v и v' съвпадат за променливи, срещащи се в τ (а в случая за $\mathtt{x} = \tau$), то последното е равно на $v'(\mathtt{x}) = [\![\mathtt{x}]\!]^{\mathbf{M}}v' = [\![\tau]\!]^{\mathbf{M}}v'$.

*Едно от често използваните в алгебрата означения за стойността на терм τ в структура $\mathbf{M}$ при оценка v е $\tau^{\mathbf{M}}[v]$. За съжаление в математическата логика се използват различни означения за стойността на терм. Означението $[\![\tau]\!]^{\mathbf{M}}v$, което използваме в този курс, е заимствано от теорията на езиците за програмиране.

Ако $\tau = \mathtt{c}$ е символ за константа, то както $[\![\tau]\!]^{\mathbf{M}}v = [\![\mathtt{c}]\!]^{\mathbf{M}}v$, така и $[\![\tau]\!]^{\mathbf{M}}v' = [\![\mathtt{c}]\!]^{\mathbf{M}}v'$ е равно на $\mathtt{c}^{\mathbf{M}}$.

Нека $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Тогава

$$[\![\tau]\!]^{\mathbf{M}}v = \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v, [\![\tau_2]\!]^{\mathbf{M}}v, \ldots, [\![\tau_n]\!]^{\mathbf{M}}v)$$
$$[\![\tau]\!]^{\mathbf{M}}v' = \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v', [\![\tau_2]\!]^{\mathbf{M}}v', \ldots, [\![\tau_n]\!]^{\mathbf{M}}v')$$

Изразите отдясно на равенствата обаче са равни, защото съгласно индукционното предположение $[\![\tau_i]\!]^{\mathbf{M}}v = [\![\tau_i]\!]^{\mathbf{M}}v'$. ∎

✓ ***Задача* 36:** Ако термът τ не съдържа нито една променлива, то за произволна структура $\mathbf{M}$ и оценки v и v' в $\mathbf{M}$, стойността на τ в структурата $\mathbf{M}$ при оценка v е равна на стойността на τ в $\mathbf{M}$ при оценка v'.

Тъй като стойността на терм зависи само от стойността на променливите, срещащи се в терма, то ще дефинираме понятието „частична оценка“.

3.2.6. Дефиниция. *Частична оценка* в структурата $\mathbf{M}$ е функция, чиято дефиниционна област съдържа само променливи, която на всяка променлива от дефиниционната си област съпоставя елемент от универсума на $\mathbf{M}$.

Разбира се, всяка оценка в $\mathbf{M}$ е и частична оценка в $\mathbf{M}$, но не всяка частична оценка в $\mathbf{M}$ е оценка в $\mathbf{M}$.

3.2.7. Дефиниция. Нека частичната оценка v в структура $\mathbf{M}$ е дефинирана за всички променливи, срещащи се в терм τ. Тогава стойността на τ в структурата $\mathbf{M}$ при частична оценка v ще дефинираме да бъде равна на стойността на τ при коя да е оценка v', която съвпада с v за променливите, срещащи се в τ. Да забележим, че съгласно твърдение 3.2.5 така дефинираната стойност не зависи от избора на v'. Стойността на τ в структура $\mathbf{M}$ при частична оценка v означаваме също както стойността при обикновена оценка: $[\![\tau]\!]^{\mathbf{M}}v$.

3.2.8. Означение. За произволни различни променливи $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на структура $\mathbf{M}$, с

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu_1, \mu_2, \ldots, \mu_n]$$

ще означаваме частичната оценка v, чиято дефиниционна област е множеството $\{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\}$ и за която $v(\mathtt{x}_i) = \mu_i$ за всяко $i \in \{1, 2, \ldots, n\}$:

$$v(\xi) = \begin{cases} \mu_1, & \text{ако } \xi = \mathtt{x}_1 \\ \mu_2, & \text{ако } \xi = \mathtt{x}_2 \\ \ldots & \\ \mu_n, & \text{ако } \xi = \mathtt{x}_n \\ \text{недефинирано}, & \text{ако } \xi \notin \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\} \end{cases}$$

3.2.9. Означение. а) Когато τ е терм, ще използваме записа

$$\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$$

когато искаме да кажем, че променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са различни по между си и всички променливи, които се срещат в τ, са измежду $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

б) Нека $\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ е терм и $\mu_1, \mu_2, \ldots, \mu_n$ са произволни елементи на универсума на структурата $\mathbf{M}$. Ще пишем

$$[\![\tau]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n]$$

вместо

$$[\![\tau]\!]^{\mathbf{M}}[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu_1, \mu_2, \ldots, \mu_n]$$

✓ ***Задача* 37:** Нека структурата $\mathbf{M}$ е с универсум множеството на реалните числа, двуместният функционален символ $\mathtt{f}$ се интерпретира като функцията събиране (т.е. $\mathtt{f}^{\mathbf{M}}(a, b) = a + b$ за произволни $a, b \in \mathbb{R}$) и двуместният функционален символ $\mathtt{g}$ се интерпретира като умножение (т.е. $\mathtt{g}^{\mathbf{M}}(a, b) = ab$ за произволни $a, b \in \mathbb{R}$). Намерете такъв терм $\tau(\mathtt{x}_1, \mathtt{x}_2)$, че

$$[\![\tau]\!]^{\mathbf{M}}[a, b] = a^2 + b$$

за произволни реални числа a и b.

3.3. Вярност на формула в структура

Да си припомним, че стойността на един терм в структура зависи от оценката, при която оценяваме терма. Това е така, защото структурата не дава стойност на променливите, които се срещат в терма. Също така да си припомним, че за да оценим един терм е достатъчно да знаем каква е стойността на променливите, които се срещат в него (твърдение 3.2.5).

Подобно се оказва и положението при формулите с тази разлика, че тук съществена е само стойността на свободните променливи. Да разгледаме например формулата

$$\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}) \tag{9}$$

За да може да се дефинира верността на тази формула е необходимо:

– структурата да каже кое е множеството, в което „работи“ кванторът $\forall \mathtt{x}$, т.е. универсумът;

– структурата да каже как се интерпретира предикатният символ $\mathtt{p}$;

– оценката да каже каква е стойността на променливите $\mathtt{y}$, $\mathtt{z}$ и $\mathtt{t}$. Стойността на останалите променливи, включително на $\mathtt{x}$, е без значение.

Нека например универсумът на структурата $\mathbf{M}$ е множеството на естествените числа и $\mathtt{p}^{\mathbf{M}}$ е предикатът

$$\mathtt{p}^{\mathbf{M}}(n, m, k, l) \longleftrightarrow m^{n+3} + m^{n+3} \neq l^{n+3}$$

В такъв случай формула (9) е вярна при оценка v тогава и само тогава, когато е вярно съждението:

За всяко естествено число n е вярно $a^{n+3} + b^{n+3} \neq c^{n+3}$

където $a = v(\mathtt{y})$, $b = v(\mathtt{z})$ и $c = v(\mathtt{t})$.

В общия случай, ако универсумът на структурата $\mathbf{M}$ е $|\mathbf{M}|$, тогава горната формула е вярна в $\mathbf{M}$ при оценка v тогава и само тогава, когато е вярно съждението:

За всеки елемент μ на $|\mathbf{M}|$ е вярно $\mathtt{p}^{\mathbf{M}}(\mu, v(\mathtt{y}), v(\mathtt{z}), v(\mathtt{t}))$.

Сега нека разгледаме по-общата ситуация на формула, в която след квантора стои не атомарна формула, а произволна формула

$$\forall \mathtt{x}\, \varphi$$

Кога трябва да считаме, че тази формула е вярна в структурата $\mathbf{M}$ при оценка v? Като пръв и най-естествен опит може да пробваме със следното съждение:

При всяка възможна стойност на $\mathtt{x}$ от универсума на $\mathbf{M}$ формулата φ е вярна.

Един проблем в тази формулировка е това, че в нея нищо не се казва за останлите променливи във φ и за оценката, която дава стойност на тези променливи. На втори опит получаваме следното:

При всяка възможна стойност на $\mathtt{x}$ *от универсума на* $\mathbf{M}$ *формулата* φ *е вярна при оценка* v*.*

В тази формулировка обаче възниква друг проблем. Какво означава „вярна при оценка v при еди-каква си стойност на променливата $\mathtt{x}$“? Та нали оценката v дава стойност на всички променливи, включително и на $\mathtt{x}$? На трети опит получаваме следното съждение:

За всеки елемент μ *на универсума на* $\mathbf{M}$ *формулата* φ *е вярна при модифицираната оценка* v'*, която се дефинира по следния начин:*

$$v'(\xi) = \begin{cases} \mu, & \text{ако } \xi = \mathtt{x} \\ v(\xi), & \text{иначе} \end{cases}$$

Именно на този вариант ще се спрем, когато дефинираме стойността на формула в структура при оценка. Преди това обаче нека дефинираме по-формално понятието „модифицирана оценка“.

✓ **3.3.1. Дефиниция.** Нека v е оценка в структурата $\mathbf{M}$, $\mathtt{x}$ е променлива и μ е елемент на универсума на $\mathbf{M}$. Тогава оценката

$$v'(\xi) = \begin{cases} \mu, & \text{ако } \xi = \mathtt{x} \\ v(\xi), & \text{иначе} \end{cases}$$

се нарича *модифицирана оценка* и ще бъде означавана така:

$$[\mathtt{x} \coloneqq \mu | v]$$

С други думи, модифицираната оценка $[\mathtt{x} \coloneqq \mu|v]$ е функция, която при аргумент $\mathtt{x}$ връща стойност μ, а за останалите аргументи съвпада с оценката v.

Използвайки модифицирани оценки, можем да дадем дефиницията за верността на формула по следния начин:

✓ **3.3.2. Дефиниция.** Нека $\mathbf{M}$ е структура. За всяка формула φ от сигнатурата на $\mathbf{M}$ и оценка v в $\mathbf{M}$ ще дефинираме съждение, което ще записваме

$$\mathbf{M} \vDash \varphi[v]$$

и ще четем така: „Формулата φ е вярна в структурата $\mathbf{M}$ при оценка v.“

Съждението $\mathbf{M} \vDash \varphi[v]$ се дефинира рекурсивно по формулата φ както следва:

а) ако $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ е атомарна формула, то $\mathbf{M} \vDash \varphi[v]$ е съждението*

$$\mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v, [\![\tau_2]\!]^{\mathbf{M}}v, \ldots, [\![\tau_n]\!]^{\mathbf{M}}v)$$

б) $\mathbf{M} \vDash (\psi_1 \,\&\, \psi_2)[v]$ е съждението

$\mathbf{M} \vDash \psi_1[v]$ *и* $\mathbf{M} \vDash \psi_2[v]$

в) $\mathbf{M} \vDash (\psi_1 \vee \psi_2)[v]$ е съждението

$\mathbf{M} \vDash \psi_1[v]$ *или* $\mathbf{M} \vDash \psi_2[v]$

г) $\mathbf{M} \vDash (\psi_1 \Rightarrow \psi_2)[v]$ е съждението

ако $\mathbf{M} \vDash \psi_1[v]$, *то* $\mathbf{M} \vDash \psi_2[v]$

д) $\mathbf{M} \vDash \neg\psi[v]$ е съждението

не е вярно, че $\mathbf{M} \vDash \psi[v]$

е) $\mathbf{M} \vDash \forall\mathtt{x}\,\psi[v]$ е съждението

за всяко $\mu \in |\mathbf{M}|$ *е вярно* $\mathbf{M} \vDash \psi[\mathtt{x} := \mu|v]$

ж) $\mathbf{M} \vDash \exists\mathtt{x}\,\psi[v]$ е съждението

съществува такова $\mu \in |\mathbf{M}|$, *че* $\mathbf{M} \vDash \psi[\mathtt{x} := \mu|v]$

3.3.3. Дефиниция. Когато съждението $\mathbf{M} \vDash \varphi[v]$ не е вярно, казваме, че формулата φ не е вярна в структурата $\mathbf{M}$ при оценка v.

✓ **3.3.4. Пример.** Нека сигнатурата $\mathfrak{sig}$ е такава, че в нея има единствен символ за константа $\mathtt{c}$, единствен функционален символ $\mathtt{f}$, който е двуместен, и единствен предикатен символ $\mathtt{p}$, който също е двуместен. Нека структурата $\mathbf{M}$ за $\mathfrak{sig}$ е с универсум множеството на реалните числа и

$$\mathtt{c}^{\mathbf{M}} = 3$$
$$\mathtt{f}^{\mathbf{M}}(x, y) = x + y$$
$$\mathtt{p}^{\mathbf{M}}(x, y) \longleftrightarrow x < y$$

Нека освен това оценката v е такава, че $v(\mathtt{x}) = 35$ и $v(\mathtt{y}) = 13$. Тогава атомарната формула $\mathtt{p}(\mathtt{f}(\mathtt{x}, \mathtt{c}), \mathtt{y})$ не е вярна в структурата $\mathbf{M}$ при

*Ако искаме, може да си припомним, че $[\![\mathtt{p}]\!]^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v, [\![\tau_2]\!]^{\mathbf{M}}v, \ldots, [\![\tau_n]\!]^{\mathbf{M}}v)$ е съкратен запис на

$$\langle [\![\tau_1]\!]^{\mathbf{M}}v, [\![\tau_2]\!]^{\mathbf{M}}v, \ldots, [\![\tau_n]\!]^{\mathbf{M}}v\rangle \in \mathtt{p}^{\mathbf{M}}$$

оценка v, защото

$$\begin{aligned}\mathbf{M} \vDash \mathtt{p}(\mathtt{f}(\mathtt{x},\mathtt{c}),\mathtt{y})[v] &\longleftrightarrow \mathtt{p}^{\mathbf{M}}(\mathtt{f}^{\mathbf{M}}(v(\mathtt{x}),\mathtt{c}^{\mathbf{M}}),v(\mathtt{y}))\\ &\longleftrightarrow \mathtt{p}^{\mathbf{M}}(\mathtt{f}^{\mathbf{M}}(35,3),13)\\ &\longleftrightarrow \mathtt{p}^{\mathbf{M}}(35+3,13)\\ &\longleftrightarrow 35+3<13 \longleftrightarrow \text{лъжа}\end{aligned}$$

✓ **3.3.5. Дефиниция.** Две формули φ и ψ са *еквивалентни*, ако при произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$, съжденията $\mathbf{M} \vDash \varphi[v]$ и $\mathbf{M} \vDash \psi[v]$ са еквивалентни. Записваме това така:

$$\vDash \varphi \Leftrightarrow \psi$$

✓ **3.3.6. Следствие.**

а) $\vDash \varphi \Leftrightarrow \varphi$

б) ако $\vDash \varphi \Leftrightarrow \psi$, *то* $\vDash \psi \Leftrightarrow \varphi$

в) ако $\vDash \varphi \Leftrightarrow \psi$ *и* $\vDash \psi \Leftrightarrow \chi$, *то* $\vDash \varphi \Leftrightarrow \chi$

Доказателство. Следва непосредствено от дефиниция 3.3.5. ■

Еквивалентните формули не са задължително равни, нито дори конгруентни. Може да си мислим за тях като изрази, които винаги имат една и съща стойност. Също както изразите

$$(x+y)^2 \text{ и } x^2+2xy+y^2$$

са различни, но винаги са равни, така и еквивалентните формули дори да са различни като запис, винаги имат еднаква вярност. Да бъдат две формули еквивалентни означава, че смисълът им е един и същ във всяка структура и при всяка оценка. Ако две формули са еквивалентни, то без значение каква е структурата и каква е оценката, винаги ако едната от тези формули се окаже вярна, то и другата ще бъде вярна, и ако едната от тях е невярна, то и другата ще бъде невярна.

3.3.7. Пример. Формулите

$$\mathtt{p}(\mathtt{x}) \vee \mathtt{q}(\mathtt{f}(\mathtt{y})) \text{ и } \mathtt{q}(\mathtt{f}(\mathtt{y})) \vee \mathtt{p}(\mathtt{x})$$

са еквивалентни — без значение как структурата интерпретира символите `p` и `f` и как оценката оценява променливите `x` и `y`, винаги ако едната от тези формули се окаже вярна, то и другата ще бъде вярна, и ако едната от тях е невярна, то и другата ще бъде невярна.

Когато в някой аритметичен израз заменим някой подизраз с друг подизраз, който е равен, тогава и целият израз ще бъде равен. Например изразът

$$\oint_C (dx + (a+b)^2\, dy)$$

е равен на израза

$$\oint_C (dx + (a^2 + 2ab + b^2)\, dy)$$

като за да стигнем до този извод дори не е нужно да знаем какво значи криволинеен интеграл.* Нещо подобно е вярно и за еквивалентните формули и трябва да си го докажем.

✓ **3.3.8. Твърдение.** *Нека формулата φ има подформула ψ и формулата ψ е еквивалентна на ψ'. Ако формулата φ' се получава от φ като заменим подформулата ψ с ψ', то формулите φ и φ' са еквивалентни.*

Доказателство. С индукция по построението на формулата φ.

Ако $\psi = \varphi$, то като заменим ψ с еквивалентна на нея формула ψ', ще получим $\varphi' = \psi'$ и значи φ и φ' ще са еквивалентни. Остава да разгледаме случая когато $\psi \neq \varphi$.

Ако φ е атомарна, то единствената ѝ подформула е $\psi = \varphi$, а вече разгледахме този случай.

Ако $\varphi = \chi_1 \,\&\, \chi_2$ и $\psi \neq \varphi$, то ψ ще е подформула на χ_1 или χ_2. Нека за определеност ψ е подформула на χ_1 и като заменим ψ на ψ' от χ_1 получаваме χ_1'. Съгласно индукционното предположение χ_1 и χ_1' са еквивалентни. Тъй като $\varphi' = \chi_1' \,\&\, \chi_2$, от тук получаваме, че φ и φ' също са еквивалентни.**

Случаите когато $\varphi = \chi_1 \vee \chi_2$ и $\varphi = \chi_1 \Rightarrow \chi_2$ се разглеждат аналогично.

Ако $\varphi = \neg\chi$ и $\psi \neq \varphi$, то ψ ще е подформула на χ. Като заменим ψ на ψ' от χ получаваме χ'. Съгласно индукционното предположение χ и

*Всъщност и двата интеграла са равни на нула за произволна затворена крива C.

**По-подробно това се обосновава така. Щом при произволна структура $\mathbf{M}$ и оценка v, $\mathbf{M} \vDash \chi_1[v]$ е еквивалентно на $\mathbf{M} \vDash \chi_1'[v]$, значи

$$\begin{aligned}
\mathbf{M} \vDash \varphi[v] &\longleftrightarrow \mathbf{M} \vDash (\chi_1 \,\&\, \chi_2)[v] \\
&\longleftrightarrow \mathbf{M} \vDash \chi_1[v] \text{ и } \mathbf{M} \vDash \chi_2[v] \\
&\longleftrightarrow \mathbf{M} \vDash \chi_1'[v] \text{ и } \mathbf{M} \vDash \chi_2[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\chi_1' \,\&\, \chi_2)[v] \\
&\longleftrightarrow \mathbf{M} \vDash \varphi'[v]
\end{aligned}$$

χ' са еквивалентни. Тъй като $\varphi' = \neg\chi'$, от тук получаваме, че φ и φ' също са еквивалентни.

Ако $\varphi = \forall \mathtt{x}\, \chi$ и $\psi \neq \varphi$, то ψ е подформула на χ. Като заменим ψ с ψ' от χ получаваме χ'. Съгласно индукционното предположение χ и χ' са еквивалентни. Тъй като $\varphi' = \forall \mathtt{x}\, \chi'$, то от тук получаваме, че φ и φ' също са еквивалентни.*

Случаят когато $\varphi = \exists \mathtt{x}\, \chi$ се разглежда аналогично. ∎

Също както за термовете определихме как да ги оценяваме при частична оценка (вж. дефиниция 3.2.7), полезно е същото да можем да правим и при формулите. По този начин ще може да използваме напр. записа

$$\mathbf{M} \vDash \varphi[5, 4]$$

когато искаме да кажем, че формулата φ е вярна в $\mathbf{M}$ когато стойността на някои, предварително уточнени променливи, е 5 и 6. Дефиницията може да бъде аналогична:

3.3.9. Дефиниция. Нека частичната оценка v в структура $\mathbf{M}$ е дефинирана за всички свободни променливи във формулата φ. Ще казваме, че φ е вярна в структурата $\mathbf{M}$ при частична оценка v, ако φ е вярна в $\mathbf{M}$ при коя да е оценка v', която съвпада с v за свободните променливи на φ. Ще означаваме това така: $\mathbf{M} \vDash \varphi[v]$.

За да бъде смислена току-що дадената дефиниция, е нужно да покажем, че верността на φ не зависи от конкретния избор на оценката v'. Това следва от следното твърдение:

✓ **3.3.10. Твърдение.** *Нека v и v' са оценки в структура $\mathbf{M}$. Ако v и v' съвпадат за всички свободни променливи на формулата φ, то $\mathbf{M} \vDash \varphi[v]$ е еквивалентно на $\mathbf{M} \vDash \varphi[v']$.*

*По-подробно това се обосновава така. Щом при произволна структура $\mathbf{M}$ и оценка v, $\mathbf{M} \vDash \chi[v]$ е еквивалентно на $\mathbf{M} \vDash \chi'[v]$, значи

$$\begin{aligned}
\mathbf{M} \vDash \varphi[v] &\longleftrightarrow \mathbf{M} \vDash \forall \mathtt{x}\, \chi[v] \\
&\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно, че } \mathbf{M} \vDash \chi[\mathtt{x} \coloneqq \mu|v] \\
&\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно, че } \mathbf{M} \vDash \chi'[\mathtt{x} \coloneqq \mu|v] \\
&\longleftrightarrow \mathbf{M} \vDash \forall \mathtt{x}\, \chi'[v] \\
&\longleftrightarrow \mathbf{M} \vDash \varphi'[v]
\end{aligned}$$

✓ Доказателство. С индукция по формулата φ. Ако φ е атомарна и има вида $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v, [\![\tau_2]\!]^{\mathbf{M}}v, \ldots, [\![\tau_n]\!]^{\mathbf{M}}v)$$

Съгласно твърдение 3.2.5, последното е еквивалентно на

$$\mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v', [\![\tau_2]\!]^{\mathbf{M}}v', \ldots, [\![\tau_n]\!]^{\mathbf{M}}v') \longleftrightarrow \mathbf{M} \vDash \varphi[v']$$

Ако $\varphi = \psi_1 \,\&\, \psi_2$, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{M} \vDash \psi_1[v] \text{ и } \mathbf{M} \vDash \psi_2[v]$$

Съгласно индукционното предположение, последното е еквивалентно на

$$\mathbf{M} \vDash \psi_1[v'] \text{ и } \mathbf{M} \vDash \psi_2[v'] \longleftrightarrow \mathbf{M} \vDash \varphi[v']$$

Когато φ има вида $\psi_1 \vee \psi_2$ или $\psi_1 \Rightarrow \psi_2$, може да разсъждаваме аналогично.

Ако $\varphi = \forall \mathtt{x}\, \psi$, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[\mathtt{x} := \mu|v]$$

и

$$\mathbf{M} \vDash \varphi[v'] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[\mathtt{x} := \mu|v']$$

Съгласно индукционното предположение, изразите отдясно на тези две еквивалентности са еквивалентни. (Защо може да прилагаме индукционното предположение?)

Когато $\varphi = \exists \mathtt{x}\, \psi$, се разсъждава аналогично. ∎

✓ **3.3.11. Означение.**

а) Когато φ е формула, ще използваме записа

$$\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$$

когато искаме да кажем, че променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са различни по между си и всички свободни променливи на φ, са измежду $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

б) Нека $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ е формула и $\mu_1, \mu_2, \ldots, \mu_n$ са произволни елементи на универсума на структурата $\mathbf{M}$. Ще пишем

$$\mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n]$$

вместо

$$\mathbf{M} \vDash \varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n := \mu_1, \mu_2, \ldots, \mu_n]$$

✓ ***Задача* 38:** Нека структурата $\mathbf{M}$ е с универсум множеството на реалните числа, двуместният функционален символ $\mathtt{f}$ се интерпретира като функцията събиране (т.е. $\mathtt{f}^{\mathbf{M}}(a,b) = a+b$ за произволни $a, b \in \mathbb{R}$), двуместният функционален символ $\mathtt{g}$ се интерпретира като умножение (т. е. $\mathtt{g}^{\mathbf{M}}(a,b) = ab$ за произволни $a, b \in \mathbb{R}$) и двуместният предикатен символ $\mathtt{p}$ се интерпретира като равенство (т.е. $\mathtt{p}^{\mathbf{M}}(a,b) \longleftrightarrow a = b$ за произволни $a, b \in \mathbb{R}$). Намерете такива терм $\tau(\mathtt{x}_1, \mathtt{x}_2)$ и атомарна формула $\varphi(\mathtt{x}_1, \mathtt{x}_2)$, че

$$[\![\tau]\!]^{\mathbf{M}}[a,b] = a^2 + b$$
$$\mathbf{M} \vDash \varphi[a,b] \longleftrightarrow a^2 = 2b$$

за произволни реални числа a и b.

Хубаво е, че сме дефинирали как се прилага субституция към произволен терм, обаче ползата от това не би била голяма, ако стойността при оценка v на терма след прилагане на субституцията не е такава, каквато трябва. Да приложим субституция s означава да заменим всяка променлива $\mathtt{x}$ с $s(\mathtt{x})$. Нека оценката w дава на променливата $\mathtt{x}$ стойност равна на стойността при оценка v на $s(\mathtt{x})$. Тогава може да очакваме, че стойността на τ при оценка w е равна на стойността на τs при първоначалната оценка v. Следващото твърдение показва, че това е така. След това, в твърдение 3.3.17, ще видим че подобно твърдение е вярно и за формулите.

3.3.12. Твърдение. *Нека s е произволна субституция, а v произволна оценка в някоя структура $\mathbf{M}$. Да дефинираме оценката w по следния начин:*

$$w(\mathtt{z}) = [\![\mathtt{z}s]\!]^{\mathbf{M}} v$$

Тогава за всеки терм τ

$$[\![\tau s]\!]^{\mathbf{M}} v = [\![\tau]\!]^{\mathbf{M}} w$$

Доказателство. С индукция по терма τ. Ако термът $\tau = \mathtt{x}$ е променлива, то получаваме исканото от дефиницията на w:

$$[\![\tau s]\!]^{\mathbf{M}} v = [\![\mathtt{x}s]\!]^{\mathbf{M}} v = w(\mathtt{x}) = [\![\tau]\!]^{\mathbf{M}} w$$

Ако термът $\tau = \mathtt{c}$ е символ за константа, то получаваме исканото съвсем лесно:

$$[\![\tau s]\!]^{\mathbf{M}} v = [\![\mathtt{c}s]\!]^{\mathbf{M}} v = [\![\mathtt{c}]\!]^{\mathbf{M}} v = \mathtt{c}^{\mathbf{M}} = [\![\tau]\!]^{\mathbf{M}} w$$

Ако $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$, то получаваме исканото, използвайки индукционното предположение за $\tau_1, \tau_2, \ldots, \tau_n$:

$$\begin{aligned}
[\![\tau s]\!]^{\mathbf{M}} v &= [\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)s]\!]^{\mathbf{M}} v \\
&= [\![\mathtt{f}(\tau_1 s, \tau_2 s, \ldots, \tau_n s)]\!]^{\mathbf{M}} v && \text{(деф. 2.7.2)} \\
&= \mathtt{f}^{\mathbf{M}}([\![\tau_1 s]\!]^{\mathbf{M}} v, [\![\tau_2 s]\!]^{\mathbf{M}} v, \ldots, [\![\tau_n s]\!]^{\mathbf{M}} v) && \text{(деф. 3.2.2 в))} \\
&= \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}} w, [\![\tau_2]\!]^{\mathbf{M}} w, \ldots, [\![\tau_n]\!]^{\mathbf{M}} w) && \text{(инд. предп.)} \\
&= [\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)]\!]^{\mathbf{M}} w && \text{(деф. 3.2.2 в))} \\
&= [\![\tau]\!]^{\mathbf{M}} w
\end{aligned}$$

■

Да си припомним, че записът $\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ означава, че променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са различни по между си и всички променливи, срещащи се в τ, са измежду $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$. Освен това, ако $\sigma_1, \sigma_2, \ldots, \sigma_n$ също са термове (с променливи не задължително измежду $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$), тогава термът $\tau[\sigma_1, \sigma_2, \ldots, \sigma_n]$ се получава като заместим в τ всяко срещане на променлива измежду $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ със съответния терм от $\sigma_1, \sigma_2, \ldots, \sigma_n$. Използвайки тези означения, може да преформулираме току-що доказаното твърдение по следния начин:

✓ **3.3.13. Твърдение (лема за субституциите за термове).** *Нека $\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n), \sigma_1, \sigma_2, \ldots, \sigma_n$ са термове и v е оценка в структурата* $\mathbf{M}$. *Тогава*

$$[\![\tau[\sigma_1, \sigma_2, \ldots, \sigma_n]]\!]^{\mathbf{M}} v = [\![\tau]\!]^{\mathbf{M}}[[\![\sigma_1]\!]^{\mathbf{M}} v, [\![\sigma_2]\!]^{\mathbf{M}} v, \ldots, [\![\sigma_n]\!]^{\mathbf{M}} v]$$

Преди да докажем аналогично твърдение за произволни формули, ще докажем като лема частния случай когато няма квантори с опасни променливи.

3.3.14. Лема. *Нека φ е формула. Ако субституцията s е такава, че φ не съдържа квантори с опасни променливи, то за произволна оценка v в структура* $\mathbf{M}$ *може да дефинираме нова оценка w*

$$w(\mathtt{z}) = [\![\mathtt{z}s]\!]^{\mathbf{M}} v$$

и, ако направим това, тогава

$$\mathbf{M} \vDash \varphi s[v] \longleftrightarrow \mathbf{M} \vDash \varphi[w]$$

Доказателство. С индукция по формулата φ. Ако $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ е атомарна, то получаваме искането, използвайки вече доказаното в твърдение 3.3.13:

$$\begin{aligned}
\mathbf{M} \vDash \varphi s[v] &\longleftrightarrow \mathbf{M} \vDash \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)s[v] \\
&\longleftrightarrow \mathbf{M} \vDash \mathtt{p}(\tau_1 s, \tau_2 s, \ldots, \tau_n s)[v] && \text{(деф. 2.7.2)} \\
&\longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1 s]\!]^{\mathbf{M}} v, [\![\tau_2 s]\!]^{\mathbf{M}} v, \ldots, [\![\tau_n s]\!]^{\mathbf{M}} v) && \text{(деф. 3.3.2 а))} \\
&\longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}} w, [\![\tau_2]\!]^{\mathbf{M}} w, \ldots, [\![\tau_n]\!]^{\mathbf{M}} w) && \text{(тв. 3.3.13)} \\
&\longleftrightarrow \mathbf{M} \vDash \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)[w] && \text{(деф. 3.3.2 а))} \\
&\longleftrightarrow \mathbf{M} \vDash \varphi[w]
\end{aligned}$$

Ако $\varphi = \bot$, то както $\mathbf{M} \vDash \varphi[w]$, така и $\mathbf{M} \vDash \varphi s[v]$ са неверни.

Ако $\varphi = (\psi \,\&\, \chi)$, то да забележим, че всяка свободна променлива в ψ или χ е свободна също и в φ, и значи всяка опасна променлива при прилагане на s към ψ или χ е опасна и при прилагане на s към φ. Следователно ако в φ няма квантори с опасни променливи, то в ψ и χ също няма да има такива квантори. В такъв случай прилагането на s е проста замяна на всяка свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$ и значи $\varphi s = \psi s \,\&\, \chi s$. Това ни дава възможност да сведем към индукционното предположение по следния начин:

$$\begin{aligned}
\mathbf{M} \vDash \varphi s[v] &\longleftrightarrow \mathbf{M} \vDash (\psi \,\&\, \chi)s[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\psi s \,\&\, \chi s)[v] && \text{(деф. 2.7.11)} \\
&\longleftrightarrow \mathbf{M} \vDash \psi s[v] \text{ и } \mathbf{M} \vDash \psi s[v] && \text{(деф. 3.3.2)} \\
&\longleftrightarrow \mathbf{M} \vDash \psi[w] \text{ и } \mathbf{M} \vDash \chi[w] && \text{(инд. предп.)} \\
&\longleftrightarrow \mathbf{M} \vDash (\psi \,\&\, \chi)[w] && \text{(деф. 3.3.2)} \\
&\longleftrightarrow \mathbf{M} \vDash \varphi[w]
\end{aligned}$$

Случаите когато $\varphi = (\psi \vee \chi)$ и $\varphi = (\psi \Rightarrow \chi)$ са аналогични.

Ако $\varphi = \forall \mathtt{x}\, \psi$, то да забележим, че свободните променливи на ψ са свободните променливи на φ плюс може би още и променливата $\mathtt{x}$. Прилагането на s към φ означава да заменим всяка свободна променлива $\mathtt{z}$ в φ с $s(\mathtt{z})$, т. е. на всяка свободна променлива $\mathtt{z}$ на ψ, която е различна от $\mathtt{x}$, с $s(\mathtt{z})$. Това ни подсказва, че всъщност става въпрос за прилагане на субституцията $[\mathtt{x} \coloneqq \mathtt{x}|s]$ към ψ. Това наистина е така, защото в ψ няма квантори с опасни променливи при тази субституция — въпреки че в ψ може да има свободна променлива $\mathtt{x}$, тази субституция „заменя“ $\mathtt{x}$ пак с $\mathtt{x}$, а съгласно дефиниция 2.7.10 такава замяна не създава опасни променливи. Значи $\varphi s = \forall \mathtt{x}\, (\psi[\mathtt{x} \coloneqq \mathtt{x}|s])$, което ни дава възможност да

разпишем лявата част на исканата еквивалентност по следния начин:

$$\begin{aligned}\mathbf{M} \vDash \varphi s[v] &\longleftrightarrow \mathbf{M} \vDash (\forall \mathtt{x}\, \psi) s[v] \\ &\longleftrightarrow \mathbf{M} \vDash (\forall \mathtt{x}\, \psi[\mathtt{x} \coloneqq \mathtt{x}|s])[v] \\ &\longleftrightarrow \text{ за всяко } \mu \in |\mathbf{M}|\ \mathbf{M} \vDash \psi[\mathtt{x} \coloneqq \mathtt{x}|s][\mathtt{x} \coloneqq \mu|v] \quad \text{(деф. 3.3.2)}\end{aligned}$$

За формулата ψ вече видяхме, че не съдържа квантори с опасни променливи при субституция $[\mathtt{x} \coloneqq \mathtt{x}|s]$. Следователно съгласно индукционното предположение изразът, до който достигнахме, е еквивалентен на

$$\text{за всяко } \mu \in |\mathbf{M}|\ \mathbf{M} \vDash \psi[w']$$

където оценката w' е такава че за произволна променлива $\mathtt{z}$:

$$w'(\mathtt{z}) = [\![\mathtt{z}[\mathtt{x} \coloneqq \mathtt{x}|s]]\!]^{\mathbf{M}}[\mathtt{x} \coloneqq \mu|v]$$

Ако разпишем и лявата страна на търсената еквивалентност, ще получим

$$\begin{aligned}\mathbf{M} \vDash \varphi[w] &\longleftrightarrow \mathbf{M} \vDash \forall \mathtt{x}\, \psi[w] \\ &\longleftrightarrow \text{ за всяко } \mu \in |\mathbf{M}|\ \mathbf{M} \vDash \psi[\mathtt{x} \coloneqq \mu|w] \qquad \text{(деф. 3.3.2)}\end{aligned}$$

Следователно остава да докажем че

$$\mathbf{M} \vDash \psi[w'] \longleftrightarrow \mathbf{M} \vDash \psi[\mathtt{x} \coloneqq \mu|w]$$

Съгласно твърдение 3.3.10 ще бъде достатъчно да установим, че оценките w' и $[\mathtt{x} \coloneqq \mu|w]$ съвпадат за всяка свободна променлива на ψ.

Нека $\mathtt{z}$ е произволна свободна променлива на ψ. Ако $\mathtt{z} = \mathtt{x}$, то тогава

$$w'(\mathtt{x}) = [\![\mathtt{x}[\mathtt{x} \coloneqq \mathtt{x}|s]]\!]^{\mathbf{M}}[\mathtt{x} \coloneqq \mu|v] = [\![\mathtt{x}]\!]^{\mathbf{M}}[\mathtt{x} \coloneqq \mu|v] = \mu = [\mathtt{x} \coloneqq \mu|w](\mathtt{x})$$

Ако пък $\mathtt{z}$ е друга свободна променлива на ψ, то

$$w'(\mathtt{z}) = [\![\mathtt{z}[\mathtt{x} \coloneqq \mathtt{x}|s]]\!]^{\mathbf{M}}[\mathtt{x} \coloneqq \mu|v] = [\![s(\mathtt{z})]\!]^{\mathbf{M}}[\mathtt{x} \coloneqq \mu|v]$$

Ако променливата $\mathtt{x}$ се срещаше в терма $s(\mathtt{z})$, то тя би била опасна променлива при прилагане на s към $\forall \mathtt{x}\, \psi$, а пък там няма квантори с опасни променливи. Следователно $\mathtt{x}$ не се среща в $s(\mathtt{z})$ и значи може да продължим горните равенства по следния начин:

$$[\![s(\mathtt{z})]\!]^{\mathbf{M}}[\mathtt{x} \coloneqq \mu|v] = [\![s(\mathtt{z})]\!]^{\mathbf{M}} v = w(\mathtt{z})$$

Случаят $\varphi = \exists \mathtt{x}\, \psi$ се разглежда аналогично. ∎

Тъй като искаме да отъждествяваме конгруентните формули, трябва да докажем, че стойностите на кои да е две конгруентни формули са еквивалентни.

✓ **3.3.15. Твърдение.** *Ако две формули са конгруентни, то те са еквивалентни.*

Доказателство. Нека $\varphi \equiv \psi$. Тогава формулите φ и ψ ще имат равен брой символи. Ще докажем, че тези формули са еквивалентни с пълна математическа индукция по броя на символите в тях.

Както обикновено, да разгледаме случаи според вида на формулите. Ако φ е атомарна, то $\varphi = \psi$, защото конгруентните формули могат да се различават само при свързаните променливи, а атомарните формули не съдържат свързани променливи. Ако $\varphi = \bot$, то положението е същото.

Ако $\varphi = \varphi' \,\&\, \varphi''$, то от дефиницията за конгруентност ще следва, че ψ има вида $\psi' \,\&\, \psi''$, където $\varphi' \equiv \psi'$ и $\varphi'' \equiv \psi''$. Съгласно индукционното предположение $\vDash \varphi' \Leftrightarrow \psi'$ и $\vDash \varphi'' \Leftrightarrow \psi''$ и значи от твърдение 3.3.8 получаваме, че $\varphi' \,\&\, \varphi'' \equiv \psi' \,\&\, \psi''$.

Случаите когато $\varphi = \varphi' \vee \varphi''$ и $\varphi = \varphi' \Rightarrow \varphi''$ се разглеждат аналогично.

Ако $\varphi = \forall \mathtt{x}\, \varphi'$, то от дефиницията за конгруентност ще следва, че ψ има вида $\forall \mathtt{y}\, \psi'$. Не можем директно да сведем към индукционното предположение, защото променливите $\mathtt{x}$ и $\mathtt{y}$ са свободни съответно в φ' и ψ, така че е възможно φ' и ψ' да не са конгруентни. Затова ще постъпим по-заобиколно. Нека $\mathtt{z}$ е променлива, която не се среща никъде — нито в φ, нито в ψ. Ако φ'' се получава от φ' като заменим всяко свободно срещане променливата $\mathtt{x}$ с $\mathtt{z}$ и аналогично ψ'' от ψ' като заменим $\mathtt{y}$ с $\mathtt{z}$, то $\varphi = \forall \mathtt{x}\, \varphi' \equiv \forall \mathtt{z}\, \varphi''$ и $\psi = \forall \mathtt{y}\, \psi' \equiv \forall \mathtt{z}\, \psi''$. Но $\varphi \equiv \psi$, значи $\forall \mathtt{z}\, \varphi'' \equiv \forall \mathtt{z}\, \psi''$, от където лема 2.6.13 ще ни даде $\varphi'' \equiv \psi''$ и значи може да приложим индукционното предположение за φ'' и ψ'' и да получим $\vDash \varphi'' \Leftrightarrow \psi''$. Съгласно твърдение 3.3.8, $\vDash \forall \mathtt{z}\, \varphi'' \Leftrightarrow \forall \mathtt{z}\, \psi''$.

Остава да установим, че $\vDash \forall \mathtt{x}\, \varphi' \Leftrightarrow \forall \mathtt{z}\, \varphi''$ и $\vDash \forall \mathtt{y}\, \psi' \Leftrightarrow \forall \mathtt{z}\, \psi''$. Ще докажем първата от тези две еквивалентности; втората се доказва аналогично. Първо, да забележим, че в φ' няма квантори с опасни променливи при субституция $[\mathtt{x} \coloneqq \mathtt{z}]$, защото единствената променлива, която може да бъде опасна е $\mathtt{z}$, а пък избрахме $\mathtt{z}$ да бъде напълно нова променлива. Следователно тази субституция се прилага към φ' просто като заменим всяко свободно срещане на $\mathtt{x}$ с $\mathtt{z}$. С други думи, $\varphi'' = \varphi'[\mathtt{x} \coloneqq \mathtt{z}]$.

Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Тогава от една

страна

$$\begin{aligned}\mathbf{M} \vDash \forall \mathtt{z}\, \varphi''[v] &\longleftrightarrow \mathbf{M} \vDash \forall \mathtt{z}\, (\varphi'[\mathtt{x} \coloneqq \mathtt{z}])[v] \\ &\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}|\ \mathbf{M} \vDash \varphi'[\mathtt{x} \coloneqq \mathtt{z}][\mathtt{z} \coloneqq \mu|v] \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(деф. 3.3.2)} \\ &\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}|\ \mathbf{M} \vDash \varphi'[\mathtt{x}, \mathtt{z} \coloneqq \mu, \mu|v] \quad \text{(лема 3.3.14)}\end{aligned}$$

От друга страна

$$\begin{aligned}\mathbf{M} \vDash \varphi[v] &\longleftrightarrow \mathbf{M} \vDash \forall \mathtt{x}\, \varphi'[v] \\ &\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}|\ \mathbf{M} \vDash \varphi'[\mathtt{x} \coloneqq \mu|v] \qquad \text{(деф. 3.3.2)}\end{aligned}$$

Тъй като променливата $\mathtt{z}$ не се среща в φ', то съгласно твърдение 3.3.10 $\mathbf{M} \vDash \varphi'[\mathtt{x}, \mathtt{z} \coloneqq \mu, \mu|v]$ е еквивалентно на $\mathbf{M} \vDash \varphi'[\mathtt{x} \coloneqq \mu|v]$.

Случаят $\varphi = \exists \mathtt{x}\, \varphi'$ се разглежда аналогично. ∎

Вече сме готови да докажем общия случай на лема 3.3.14.

3.3.16. Твърдение. *Нека φ е формула, s е субституция и v е оценка в структура $\mathbf{M}$. Ако оценката w е такава, че за всяка променлива $\mathtt{z}$*

$$w(\mathtt{z}) = [\![\mathtt{z}s]\!]^{\mathbf{M}} v$$

то

$$\mathbf{M} \vDash \varphi s[v] \longleftrightarrow \mathbf{M} \vDash \varphi[w]$$

Доказателство. Нека ψ е конгруентна с φ формула, която не съдържа квантори с опасни при субституция s променливи (такава формула има съгласно лема 2.6.11). Съгласно лема 3.3.14, $\mathbf{M} \vDash \psi[w]$ е еквивалентно на $\mathbf{M} \vDash (\psi s)[v]$. От тук получаваме исканото, защото формулата φ е конгруентна, а значи и еквивалентна с ψ, а от твърдение 2.6.9 имаме, че формулата φs е конгруентна, а значи и еквивалентна с формулата ψs. ∎

Използвайки други означения, може да преформулираме това твърдение по следния начин:

✓ **3.3.17. Твърдение (лема за субституциите за формули).** *Нека $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ е формула, $\tau_1, \tau_2, \ldots, \tau_n$ са термове и v е оценка в структурата $\mathbf{M}$. Тогава*

$$\mathbf{M} \vDash \varphi[\tau_1, \tau_2, \ldots, \tau_n][v] \longleftrightarrow \mathbf{M} \vDash \varphi[[\![\tau_1]\!]^{\mathbf{M}} v, [\![\tau_1]\!]^{\mathbf{M}} v, \ldots, [\![\tau_n]\!]^{\mathbf{M}} v]$$

3.4. Тъждествена вярност, изпълнимост, следване

Обикновено когато пишем някоя формула ние не си мислим, че променливите в нея имат някакви конкретни стойности. Например формулата

$$\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$$

ни казва, че двуместната функция, означена с „+“, е комутативна, т.е. $\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$ при произволни стойности на $\mathtt{x}$ и $\mathtt{y}$. Това показва, че понятието „вярност в структура при оценка“, което вече дефинирахме, не ни дава точно това, което искаме. Например горната формула ще бъде вярна в някоя структура при определена оценка тогава и само тогава, когато $\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$ е вярно при някакви конкретни стойности на променливите $\mathtt{x}$ и $\mathtt{y}$, определени от оценката, а не при произволни стойности.*

Когато ни е дадена само структура, но не и оценка, една формула може да има различни „степени на вярност“. Например:

- формулата може да бъде вярна при всяка оценка;
- формулата може да бъде вярна при поне една оценка;
- формулата може да бъде невярна при произволна оценка.

Когато за една формула кажем просто, че е вярна, без да уточним при коя оценка, удобно е да считаме, че формулата е вярна при всяка оценка. Затова можем да дадем следната дефиниция:

✓ **3.4.1. Дефиниция.** а) Формулата φ е *(тъждествено) вярна* в структура $\mathbf{M}$, ако е вярна в структурата $\mathbf{M}$ при произволна оценка v.** Записваме това така:

$$\mathbf{M} \vDash \psi$$

б) Формулата φ е *тъждествено вярна* или *(предикатна) тавтология*, ако φ е тъждествено вярна във всяка структура.*** Записваме това така:

$$\vDash \psi$$

*Нека структурата е с универсум положителните реални числа и функционалният символ „+“ е интерпретиран като операцията деление. Тогава горната формула не е вярна, защото например $2/3 \neq 3/2$. Ако обаче оценката е такава, че $v(\mathtt{x}) = v(\mathtt{y}) = 42$, тогава горната формула ще бъде вярна при тази оценка, защото $42/42 = 42/42$.

**Други термини, които означават същото като „тъждествено вярна“ са *общовалидна* и *валидна*. На английски universally valid и valid.

***За предикатните тавтологии се използват още термините *общовалидна формула* и *валидна формула*.

Забележка: Без опасност от недоразумения може да изпускаме думата „тъждествено“ от термина „тъждествено вярна формула в структура“. Причината за това е следната: съгласно дефиниция 3.3.2 сме дефинирали какво значи „вярна формула в структура **при оценка**“. Следователно ако просто кажем, че някоя формула е вярна в структура без да уточним коя е оценката, няма как да имаме предвид дефиниция 3.3.2. Най-естествено е да считаме, че в такъв случай формулата е вярна при произволна оценка, т.е. че е тъждествено вярна. Да си припомним аксиомите на моноид и полупръстен (вж. стр. 91 и 93). Ако за тези аксиоми кажем, че са верни във всеки моноид или съотв. полупръстен, ние всъщност ще имаме предвид, че тези аксиоми са тъждествено верни, а не че са верни при някоя конкретна, неясно каква оценка.

Понякога в контекста на предикатната логика думата „тавтология“ се използва само за онези предикатни тавтологии, които могат да се установят от съждителната логика. Тук няма да използваме такъв тип тавтологии и затова може да си позволим да изпускаме думата „предикатна“ от словосъчетанието „предикатна тавтология“.

Една формула може да бъде тъждествено вярна в една структура без да е тъждествено вярна във всяка структура. Например формулата $\mathtt{x.y} = \mathtt{y.x}$ е тъждествено вярна във всяка структура, която е абелева група или комутативен пръстен, но не е тъждествено вярна структурите, които са неабелеви групи или пък некомутативни пръстени. Разбира се, тази формула е и пример на формула, която може да бъде тъждествено вярна в някоя конкретна структура (коя да е абелева група или комутативен пръстен) без да бъде тъждествено вярна по принцип (т.е. без да бъде предикатна тавтология).

Съгласно дефиниция 3.3.5, ако две формули φ и ψ са еквивалентни, записваме това така: $\models \varphi \Leftrightarrow \psi$. Съгласно дефиниция 3.4.1 б) пък, когато запишем $\models \varphi \Leftrightarrow \psi$, това означава, че формулата $\varphi \Leftrightarrow \psi$ е тавтология. Следното твърдение показва, че нямаме опасна колизия в означенията, защото тези две неща са еквивалентни.

3.4.2. Твърдение. *Две формули φ и ψ са еквивалентни тогава и само тогава, когато формулата $\varphi \Leftrightarrow \psi$ е предикатна тавтология.*

Доказателство. Да си припомним, че формулата $\varphi \Leftrightarrow \psi$ е съкращение на формулата $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \varphi)$. Затова от дефиниции 3.3.5 и 3.4.1 а) следва, че φ и ψ са еквивалентни тогава и само тогава, когато при произволна структура $\mathbf{M}$ и оценка v в $\mathbf{M}$, от $\mathbf{M} \models \varphi[v]$ следва $\mathbf{M} \models \psi[v]$ и от $\mathbf{M} \models \psi[v]$ следва $\mathbf{M} \models \varphi[v]$. Последното означава, че $\mathbf{M} \models \varphi[v]$ и $\mathbf{M} \models \psi[v]$ са еквивалентни. ∎

Когато някоя формула е вярна не при всяка оценка, а при някоя оценка, или не във всяка структура, а в някоя структура, използваме термина „изпълнима формула“.

✓ **3.4.3. Дефиниция.** а) Формулата φ е *изпълнима* в структурата $\mathbf{M}$, ако съществува оценка, при която φ е вярна в $\mathbf{M}$.

б) Формулата φ е *изпълнима*, ако съществува структура, в която φ е тъждествено вярна.

в) Множество Γ от формули е *изпълнимо*, ако съществува структура, в която са тъждествено верни всички формули от Γ.

Забележка: За съжаление използваната терминология на български е малко объркваща. Вместо „изпълнима“ на английски се използва прилагателното satisfiable. На това прилагателно съответства глаголът satisfy, който на български превеждаме с напълно различна дума — „удовлетворява“. Вместо „удовлетворима формула“ казваме „изпълнима формула“ и вместо (напр. в контекста на логическото програмиране) да казваме, че дадена цел се изпълнява, казваме, че целта се удовлетворява. На английски обаче във всеки един от тези случи се използва една и съща дума.

✓ **3.4.4. Дефиниция.**

а) Формула φ е *тъждествено невярна* в структурата $\mathbf{M}$, ако при всяка оценка v в $\mathbf{M}$, формулата φ е невярна в $\mathbf{M}$ при оценка v.

б) Формулата φ е *неизпълнима* в структурата $\mathbf{M}$, ако не съществува оценка, при която φ е вярна в $\mathbf{M}$.

в) Формула φ е *тъждествено невярна*, ако при всяка структура $\mathbf{M}$ и оценка v в $\mathbf{M}$, формулата φ е невярна в $\mathbf{M}$ при оценка v.

г) Формулата φ е *неизпълнима*, ако не съществува структура, в която φ е тъждествено вярна.

Следващите две задачи показват, че макар понятията от току-що дадената дефиниция да са естествени, те по същество не ни дават нищо ново.

✓ ***Задача* 39:** Нека φ е формула и $\mathbf{M}$ е структура. Докажете, че следните три неща са еквивалентни:

- φ не е изпълнима в $\mathbf{M}$;
- φ е неизпълнима в $\mathbf{M}$;
- φ е тъждествено невярна в $\mathbf{M}$.

✓ ***Задача* 40:** Нека φ е формула. Докажете, че

- φ е неизпълнима тогава и само тогава, когато φ не е изпълнима;
- φ е тъждествено невярна тогава и само тогава, когато $\neg\varphi$ е тъждествено вярна.

✓ ***Задача* 41:** Възможно ли е една формула да не е тъждествено вярна в някоя структура, а пък да бъде изпълнима в същата структура?

***Задача* 42:** Възможно ли е една формула да бъде едновременно неизпълнима и тъждествено вярна в някоя структура? Защо при отговора на този въпрос се налага да използваме това, че универсумът на структурите е непразно множество?

✓ **3.4.5. Дефиниция.** Ако формула φ е тъждествено вярна в структура $\mathbf{M}$ (т.е. $\mathbf{M} \vDash \varphi$), казваме, че $\mathbf{M}$ е модел за φ. Ако всеки елемент на множество Γ от формули е тъждествено верен в структура $\mathbf{M}$, казваме, че $\mathbf{M}$ е модел за Γ и пишем $\mathbf{M} \vDash \Gamma$.

3.4.6. *Забележка*: Понякога в текстове, които не са писани от логици, думата „модел“ се приема за синоним на „структура“. Това е неправилно. Когато кажем, че $\mathbf{M}$ е модел, това означава не само, че $\mathbf{M}$ е структура, но също и че определени (известни може би от контекста) неща са верни в $\mathbf{M}$.

***Задача* 43:** Докажете, че ако някое множество от формули има поне един модел, то то има безброй много модели.

✓ **3.4.7. Дефиниция.** *Затворена формула* означава формула, която не съдържа свободни променливи.

3.4.8. Съгласно твърдение 3.3.10 верността на една затворена формула в структура $\mathbf{M}$ не зависи по никакъв начин от оценката, защото всеки две оценки съвпадат за свободните променливи на една затворена формула. Следователно има смисъл когато говорим за затворена формула да четем „$\mathbf{M} \vDash \varphi$“ като „$\varphi$ е вярна в $\mathbf{M}$“, изпускайки наречието „тъждествено“.

✓ **3.4.9. Твърдение.** *Ако φ е затворена формула, а $\mathbf{M}$ — произволна структура, то за всяка оценка v в $\mathbf{M}$*

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{M} \vDash \varphi \longleftrightarrow \varphi \textit{ е изпълнима в } \mathbf{M}$$

✓ Доказателство. Да припомним, че $\mathbf{M} \vDash \varphi$ означава, че формулата φ е тъждествено вярна в $\mathbf{M}$, т.е. φ е вярна при всяка оценка, а φ да бъде изпълнима в $\mathbf{M}$ означава, че формулата φ е вярна в $\mathbf{M}$ при някоя оценка.

Съгласно твърдение 3.3.10 за кои да е две оценки v_1 и v_2 в $\mathbf{M}$ съждението $\mathbf{M} \vDash \varphi[v_1]$ е еквивалентно на $\mathbf{M} \vDash \varphi[v_2]$. Следователно φ е изпълнима в $\mathbf{M}$ тогава и само тогава, когато $\mathbf{M} \vDash \varphi[v]$ е истина при някоя оценка v, тогава и само тогава, когато $\mathbf{M} \vDash \varphi[v]$ е истина за всяка оценка v, тогава и само тогава, когато $\mathbf{M} \vDash \varphi$. ■

***Задача* 44:** Възможно ли е една затворена формула да бъде изпълнима, но да не бъде тъждествено вярна?

Полезно е едно понятие, което казва, че от някакви формули следва някоя формула. Нека например множеството Γ съдържа аксиомите за полупръстен. Може да считаме, че формулата φ следва от Γ, ако φ е вярна във всички полупръстени. Следната дефиниция формализира интуицията на тази забележка:

✓ **3.4.10. Дефиниция.** Нека Γ е множество от формули. Казваме, че формулата φ *следва* от Γ, ако φ е тъждествено вярна във всяка структура, в която са тъждествено верни формулите в Γ. В този случай казваме още и накратко „от Γ следва φ“ и използваме следното означение:

$$\Gamma \vDash \varphi$$

3.4.11. *Забележка*: Нека не се объркваме и обърнем внимание, че използваме знака $\vDash$ за няколко различни цели. Когато Γ е множество от формули, тогава $\Gamma \vDash \varphi$ означава „от формулите в Γ следва φ“. Когато $\mathbf{M}$ е структура, $\mathbf{M} \vDash \varphi$ означава „φ е (тъждествено) вярно в $\mathbf{M}$“. А когато $\mathbf{M}$ е структура и v е оценка, $\mathbf{M} \vDash \varphi[v]$ означава „φ е вярна в $\mathbf{M}$ при оценка v“.

✓ ***Задача* 45:** Нека Γ и Δ са множества от формули и φ е формула. Докажете, че:

а) $\varphi \in \Gamma \longrightarrow \Gamma \vDash \varphi$

б) $\Delta \subseteq \Gamma$ и $\Delta \vDash \varphi \longrightarrow \Gamma \vDash \varphi$

в) Ако всеки елемент на Δ следва от Γ и $\Delta \vDash \varphi$, то $\Gamma \vDash \varphi$.

***Забележка*:** Условия а) и в) от задача 45 аксиоматизират това, което в абстрактната алгебрична логика се нарича *релация на следване*.*

*На английски consequence relation.

3.5. Хомоморфизми

Когато наблюдаваното поведение на една физическа система не се променя, ако подложим системата на някаква трансформация, физиците казват, че е налице *симетрия*. Симетриите са фундаментално понятие в съвременната физика и не малко от т. н. природни закони могат да се изведат като следствие от наличието на една или друга симетрия. Ето някои примери:

– **Транслация.** Всяка проста транслация в пространството е пример за симетрия. Да си изберем някаква четиримерна координатна система и да опишем поведението на света, използвайки координати от така избраната координатна система. Сега да изберем нова координатна система, която се получава от първата с транслация в пространството. Дали ако опишем поведението на света в новата координатна система ще получим нов вид физически зависимости? Не, физическите зависимости не се променят при транслация на координатната система. От тук, впрочем, може да се изведе като следствие т. н. закон за запазване на импулса.[*]

– **Ротация.** Всяка проста ротация също е пример за симетрия. Да си изберем някаква четиримерна координатна система и да опишем поведението на света, използвайки координати от така избраната координатна система. Сега да изберем нова координатна система, която има същото начало като първата, но е завъртяна. Дали ако опишем поведението на света в новата координатна система ще получим нов вид физически зависимости? Не, физическите зависимости не се променят при ротация на координатната система. От тук, впрочем, може да се изведе като следствие т. н. закон за запазване на въртящия момент.[**]

– **Отместване във времето.** Отместването във времето също е пример за симетрия. Да си изберем някаква четиримерна координатна система и да опишем поведението на света, използвайки координати от така избраната координатна система. Сега да изберем нова координатна система, която пространствено е ориентирана по същия начин като първата, но началото ѝ е отместено

[*] На земята може да се придвижваме пеш или с автомобил без да се налага да изхвърляме вещество. В космоса това не е възможно и единственото средство за придвижване е ракетният двигател.

[**] На земята може да се въртим във всички посоки, защото имаме опорна точка. В космоса телата не могат да ускорят или забавят въртенето си без да изхвърлят част от себе си. Впрочем това не е задължително да става с ракетен двигател.

във времето. Дали ако опишем поведението на света в новата координатна система ще получим нов вид физически зависимости? Не, физическите зависимости не се променят при отместване във времето — каквито са били вчера, такива ще бъдат и утре. От тук, впрочем, може да се изведе като следствие т. н. закон за запазване на енергията.

– **Скоростта на светлината.** Скоростта на светлината винаги се оказва една и съща — без значение с каква скорост се движи фенера и с каква скорост се движи измервателя. Използвайки това свойство заедно с някои други симетрии може да изведем като следствие *Специалната теория на относителността*.

– **Гладки трансформации.** Ако приемем, че свойствата на физическите обекти не се променят, ако извършим гладко преобразуване на координатите, то от тук, използвайки и някои други симетрии, може да изведем като следствие *Общата теория на относителността*.

Това, което физиците наричат симетрия, математиците наричат *автоморфизъм*. Трансформациите, при които определени свойства на системата се запазват, са от изключителна важност не само във физиката, но и в математиката. Ето някои примери:

– В линейната алгебра *линейните трансформации* запазват линейните операции: $h(\vec{x}+\vec{y}) = h(\vec{x}) + h(\vec{y})$, $h(a\vec{x}) = ah(\vec{x})$. Когато едно такова изображение е биективно, то се нарича *изоморфизъм*.

– В абстрактната алгебра *хомоморфизмите* запазват операциите в алгебричната структура. Например за хомоморфизмите между полета е вярно, че $h(0) = 0$, $h(1) = 1$, $h(x+y) = h(x) + h(y)$ и $h(xy) = h(x)h(y)$. Когато едно такова изображение е биективно, то се нарича *изоморфизъм*.

– В топологията *непрекъснатите изображения* запазват свойството „допиране“ — ако две множества A и B се допират, то $h(A)$ и $h(B)$ също ще се допират. Когато едно такова изображение е биективно, то се нарича *хомеоморфизъм*.

– В геометрията *гладките функции* запазват гладкостта — ако множеството A е гладко многообразие, то $h(A)$ също ще бъде гладко многообразие (локално). Когато едно такова изображение е биективно, то се нарича *дифеоморфизъм*.

Впрочем също както във физиката е възможно да извеждаме свойствата на физическите системи като следствие от наличието на различни

симетрии, така и в математиката е възможно да описваме математическите обекти, използвайки най-вече трансформации, запазващи свойствата им. Това е подходът, използван в *Теория на категориите*.*

Вече видяхме, че алгебричните структури се оказват частни случаи на нашето понятие за структура от дефиниция 3.1.5 (вж. напр. моноидите и полупръстените от раздел 2.5 и пример 3.1.7). Не може ли след като разгледаме как алгебриците са дефинирали понятието „хомоморфизъм“ за своите структури, от тук по аналогия да получим и дефиниция за хомоморфизъм за нашите структури? Нека видим как изглежда дефиницията на хомоморфизъм между различни видове алгебрични структури.

Езикът на групите съдържа константа e — единичния елемент — и „умножение“, означавано с точка. Ако $\mathbf{G}$ и $\mathbf{H}$ са групи, то $h\colon \mathbf{G} \to \mathbf{H}$ е хомоморфизъм, ако h е такова изображение от универсума на $\mathbf{G}$ в универсума на $\mathbf{H}$, че

$$h(e^{\mathbf{G}}) = e^{\mathbf{H}}$$

където с $e^{\mathbf{G}}$ и $e^{\mathbf{H}}$ сме означили единичните елементи съответно на $\mathbf{G}$ и $\mathbf{H}$, и освен това за произволни елементи a и b на универсума на $\mathbf{G}$

$$h(a \mathbin{.^{\mathbf{G}}} b) = h(a) \mathbin{.^{\mathbf{H}}} h(b)$$

където с $.^{\mathbf{G}}$ и $.^{\mathbf{H}}$ сме означили умноженията съответно в групите $\mathbf{G}$ и $\mathbf{H}$.

Да разгледаме сега понятието хомоморфизъм между пръстени (хомоморфизмите между полета се дефинират аналогично). Езикът на пръстените съдържа константи 0 и 1, адитивна операция „+“ и мултипликативна операция „.“. Ако $\mathbf{R}$ и $\mathbf{T}$ са пръстени, то $h\colon \mathbf{R} \to \mathbf{T}$ е хомоморфизъм, ако h е такова изображение от универсума на $\mathbf{R}$ в универсума на $\mathbf{T}$, че

$$h(0^{\mathbf{R}}) = 0^{\mathbf{T}}$$
$$h(1^{\mathbf{R}}) = 1^{\mathbf{T}}$$

*Категориите са измислени в хомологичната алгебра и в тази област те са незаменими. Освен това те са получили приложения и в много други области. Например приложенията им в компютърните науки са многобройни и разнообразни и при това важни за практиката. Категориите позволяват да даваме дефиниции на абстрактни понятия, които трудно бихме могли да формулираме без използването на теоретикокатегорен език. За съжаление нерядко тези абстракции са самоцелни и ненужни и това несправедливо е довело до лоша слава на теория на категориите.

където с $0^{\mathbf{R}}$, $0^{\mathbf{T}}$, $1^{\mathbf{R}}$ и $1^{\mathbf{T}}$ сме означили нулевия и единичния елемент на $\mathbf{R}$ и $\mathbf{T}$, и освен това за произволни елементи a и b на универсума на $\mathbf{R}$

$$h(a \ +^{\mathbf{R}} \ b) = h(a) \ +^{\mathbf{T}} \ h(b)$$
$$h(a \ .^{\mathbf{R}} \ b) = h(a) \ .^{\mathbf{T}} \ h(b)$$

където с $+^{\mathbf{R}}$, $+^{\mathbf{T}}$, $.^{\mathbf{R}}$ и $.^{\mathbf{T}}$ сме означили адитивната и мултипликативната операция в пръстените $\mathbf{R}$ и $\mathbf{T}$.

Очевидно е, че тези дефиниции си приличат. Ако решим да си направим труда да проверим как изглеждат дефинициите на други видове хомоморфизми между алгебрични структури, ще установим, че и техните дефиниции са подобни на току-що дадените. По-точно, ще забележим, че за всички константи от езика на съответната структура искаме равенства от вида

$$h(e^{\mathbf{G}}) = e^{\mathbf{H}}$$
$$h(0^{\mathbf{R}}) = 0^{\mathbf{T}}$$
$$h(1^{\mathbf{R}}) = 1^{\mathbf{T}}$$

Това означава, че навярно ще бъде разумно в нашата дефиниция за хомоморфизъм $h\colon \mathbf{M} \to \mathbf{K}$ да поискаме за всеки символ за константа $\mathtt{c}$ от сигнатурата да се изпълнява следното равенство:

$$h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}}$$

Освен това в различните алгебрични дефиниции за хомоморфизъм се забелязва, че за всички алгебрични операции се изискват равенства от вида

$$h(a \ .^{\mathbf{G}} \ b) = h(a) \ .^{\mathbf{H}} \ h(b)$$
$$h(a \ +^{\mathbf{R}} \ b) = h(a) \ +^{\mathbf{T}} \ h(b)$$
$$h(a \ .^{\mathbf{R}} \ b) = h(a) \ .^{\mathbf{T}} \ h(b)$$

Това ни подсказва, че навярно ще бъде разумно в нашата дефиниция за хомоморфизъм $h\colon \mathbf{M} \to \mathbf{K}$ да поискаме за всеки n-местен функционален символ $\mathtt{f}$ от сигнатурата и произволни елементи $\mu_1, \mu_2, \ldots, \mu_n$ от универсума на $\mathbf{M}$ да се изпълнява следното равенство:

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = \mathtt{f}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

Остава да видим какво изискване да поискаме за предикатните символи. Алгебрата тук не може да ни помогне, защото в алгебричните

структури единственият предикатен символ е равенството. За съжаление, оказва се, че има различни смислени начини да определим кои свойства на предикатните символи трябва да се запазват. В следващата дефиниция за хомоморфизъм сме поискали от хомоморфизма да запазва възможно най-малко свойства:

3.5.1. Дефиниция. Нека $\mathbf{M}$ и $\mathbf{K}$ са произволни структури.* Казваме, че h е *хомоморфизъм* от $\mathbf{M}$ в $\mathbf{K}$ и записваме това така:

$$h\colon \mathbf{M} \to \mathbf{K}$$

ако h е изображение от универсума на $\mathbf{M}$ в универсума на $\mathbf{K}$ и освен това:

а) за всеки символ за константа $\mathtt{c}$ от сигнатурата

$$h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}}$$

б) за всеки n-местен функционален символ $\mathtt{f}$ от сигнатурата и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = \mathtt{f}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

в) за всеки n-местен предикатен символ $\mathtt{p}$ от сигнатурата и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$\text{ако } \mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n), \text{ то } \mathtt{p}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

3.5.2. Пример. Хомоморфизмите в алгебрата се оказват частни случаи на току-що даденото понятие. При символите за константи и функционалните символи сме се погрижили нашата дефиниция да съвпада с това, което се казва в алгебричните дефиниции. В нашата дефиниция обаче има допълнително изискване за предикатните символи, което го няма в алгебрата. Да видим какво казва това изискване.

Единственият предикатен символ в алгебричните структури е равенството. За равенството условието от дефиниция 3.5.1 в) казва следното: за произволни елементи μ_1 и μ_2 на универсума на $\mathbf{M}$

$$\text{ако } \mu_1 = \mu_2, \text{ то } h(\mu_1) = h(\mu_2)$$

Това обаче е очевидно — ясно е, че ако $\mu_1 = \mu_2$, то също и $h(\mu_1) = h(\mu_2)$. Следователно едно изображение е хомоморфизъм между моноиди, групи, попупръстени, пръстени, полета и т. н. тогава и само тогава, когато то е хомоморфизъм по смисъла на дефиниция 3.5.1.

*Както обикновено, когато не уточняваме с какви сигнатури работим, се предполага, че всички структури, термове и формули използват една и съща сигнатура.

В теория на графите хомоморфизмите се дефинират по следния начин: h е хомоморфизъм от граф $\mathbf{G}_1$ в граф $\mathbf{G}_2$, ако h изобразява всеки връх на $\mathbf{G}_1$ в някой връх на $\mathbf{G}_2$ и винаги когато в $\mathbf{G}_1$ има ребро от v_1 до v_2, в $\mathbf{G}_2$ ще има ребро от $h(v_1)$ до $h(v_2)$. Например ако графът $\mathbf{G}_2$ има два върха и нито едно ребро, то тогава съществува хомоморфизъм от $\mathbf{G}_1$ към $\mathbf{G}_2$ тогава и само тогава, когато върховете на $\mathbf{G}_1$ могат да се оцветят в два цвята, така че никое ребро да не свързва едноцветни върхове, т.е. когато графът е *двуделен*.

3.5.3. Пример. Да видим какво казва нашата дефиниция за хомоморфизъм в случая, когато структурите са графи. Да си припомним от пример 3.1.8, че един граф $\mathbf{G}$ може да се мисли като структура за сигнатура, в която няма символи за константи и функционални символи и има един единствено предикатен символ $\mathtt{p}$ и той е двуместен. Универсумът $|\mathbf{G}|$ на структурата се състои от върховете на графа, а $\mathtt{p}^{\mathbf{G}}(v_1, v_2)$ е истина тогава и само тогава, когато в графа има ребро от връх v_1 до връх v_2.

Тъй като при графите няма нито символи за константи, нито функционални символи, то h е хомоморфизъм от граф $\mathbf{G}_1$ в граф $\mathbf{G}_2$, ако h изобразява всеки връх на $\mathbf{G}_1$ в някой връх на $\mathbf{G}_2$ и винаги когато $\mathtt{p}^{\mathbf{G}_1}(v_1, v_2)$ е истина, т.е. когато в $\mathbf{G}_1$ има ребро от v_1 до v_2, то $\mathtt{p}^{\mathbf{G}_2}(h(v_1), h(v_2))$ също ще бъде истина, т.е. в $\mathbf{G}_2$ ще има ребро от $h(v_1)$ до $h(v_2)$. Но това е точно дефиницията от теория на графите! Следователно хомоморфизмите в теория на графите също се оказват частен случай на нашето понятие.

Казахме, че дефиниция 3.5.1 е дадена така, че от хомоморфизмите да искаме да запазват възможно най-малко свойства. Да помислим какво в повече бихме могли да поискаме от един хомоморфизъм да запазва по отношение на символите за константи и функционалните символи. Вероятно едно от най-силните условия е следното: ако $h\colon \mathbf{M} \to \mathbf{K}$ е хомоморфизъм и τ е някакъв израз, съставен от символи за константи и функционални символи (т.е. терм), то образът на стойността на τ в $\mathbf{M}$ е равен на стойността на τ в $\mathbf{K}$. Оказва се обаче, че няма нужда изрично да поискаме такова свойство в дефиницията на хомоморфизъм, защото то може да се получи като следствие. Всъщност това е причината, поради която алгебриците нямат онези проблеми, които имат логиците — дори когато алгабриците поискат в своите дефиниции h да удовлетворява възможно най-слабите условия, пак ще се окаже, че h удовлетворява всичко, каквото можем да поискаме от един хомоморфизъм. Следващото твърдение показва точно това.

(Припомнете си означение 3.2.9 б).)

3.5.4. Твърдение. *Нека* $h\colon \mathbf{M} \to \mathbf{K}$ *е хомоморфизъм и* $\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ *е терм. Тогава за произволни* $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$

$$h([\![\tau]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n]) = [\![\tau]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Доказателство. С индукция по броя на символите в терма τ. Когато $\tau = \mathtt{x}$ е променлива, то $\tau = \mathtt{x}_i$ за някое i и значи

$$h([\![\tau]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n]) = h(\mu_i) = [\![\tau]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Когато $\tau = \mathtt{c}$ е символ за константа

$$h([\![\tau]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n]) = h(\mathtt{c}^{\mathbf{M}})$$

Тъй като h е хомоморфизъм, то $h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}}$ и значи може да продължим горното равенство по следния начин:

$$h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}} = [\![\tau]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Когато $\tau = \mathtt{f}(\tau_1, \ldots, \tau_m)$

$$\begin{aligned} &h([\![\tau]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n]) \\ &\qquad = h(\mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n], \ldots, [\![\tau_m]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n])) \end{aligned}$$

Тъй като h е хомоморфизъм, последното е равно на

$$\mathtt{f}^{\mathbf{K}}(h([\![\tau_1]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n]), \ldots, h([\![\tau_m]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_n]))$$

което пък съгласно индукционното предположение е равно на

$$\mathtt{f}^{\mathbf{K}}([\![\tau_1]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)], \ldots, [\![\tau_m]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)])$$

По дефиниция 3.2.2 в), това е равно на $[\![\tau]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$. ∎

За съжаление не може да формулираме аналогично твърдение за формули. Без доказателство ще посочим, че е вярно следното значително по-слабо твърдение:

Твърдение. *Нека* $h\colon \mathbf{M} \to \mathbf{K}$ *е хомоморфизъм и* $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ *е безквантορна формула, в която не се срещат други свързителни операции, освен* & *и* $\vee$. *Тогава за произволни* $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$

ако $\mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n]$, *то* $\mathbf{K} \vDash \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$

Дефиницията на силен хомоморфизъм е аналогична на дефиницията на хомоморфизъм, но в точка 3.5.1 в) използваме еквиваленция вместо импликация:

3.5.5. Дефиниция. Нека $\mathbf{M}$ и $\mathbf{K}$ са произволни структури. Казваме, че h е *силен хомоморфизъм* от $\mathbf{M}$ в $\mathbf{K}$ и записваме това така:

$$h\colon \mathbf{M} \to \mathbf{K}$$

ако h е изображение от универсума на $\mathbf{M}$ в универсума на $\mathbf{K}$ и освен това:

а) за всеки символ за константа $\mathtt{c}$

$$h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}}$$

б) за всеки n-местен функционален символ $\mathtt{f}$ и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = \mathtt{f}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

в) за всеки n-местен предикатен символ $\mathtt{p}$ и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$\mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n) \longleftrightarrow \mathtt{p}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

3.5.6. Да видим какъв е смисълът на силните хомоморфизми в алгебрата. Тъй като в алгебричните структури единствен предикатен символ е равенството, то изображението $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм тогава и само тогава, когато h е хомоморфизъм и освен това за произволни елементи μ_1 и μ_2 на универсума на $\mathbf{M}$ е вярно, че

$$\mu_1 = \mu_2 \longleftrightarrow h(\mu_1) = h(\mu_2)$$

Посоката отляво надясно е тривиално вярна. Обратната посока обаче казва, че h е инекция. Оказва се, че това е изключително ограничително изискване, което намалява полезността на това понятие в алгебрата. Всъщност не само между алгебрични структури, но изобщо винаги когато h е силен хомоморфизъм между структури с равенство, h трябва да бъде инекция. За щастие на нас ще ни се наложи да използваме силни хомоморфизми само между структури, в които няма предикатен символ, който се интерпретира като равенство, а в този случай може да има и по-нетривиални примери за силни хомоморфизми.

Когато използваме силни хомоморфизми, може да докажем за формулите твърдение, аналогично на твърдение 3.5.4. То не е вярно за произволни формули, а само за безкванторни формули, но и това е нещо...

3.5.7. Твърдение. *Нека* $h\colon \mathbf{M} \to \mathbf{K}$ *е силен хомоморфизъм и формулата* $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ *е безкванторна. Тогава за произволни* $\mu_1, \mu_2, \ldots, \mu_n$ *от универсума на* $\mathbf{M}$

$$\mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \mathbf{K} \vDash \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Доказателство. С индукция по броя на символите в φ. Когато $\varphi = \mathtt{p}(\tau_1, \ldots, \tau_m)$ е атомарна

$$\begin{aligned}\mathbf{M} \vDash \mathtt{p}(\tau_1, \ldots, \tau_m)[\mu_1, \mu_2, \ldots, \mu_n] &\longleftrightarrow \\ &\longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_m], \ldots, [\![\tau_m]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_m])\end{aligned}$$

Тъй като h е силен хомоморфизъм, последното е еквивалентно на

$$\mathtt{p}^{\mathbf{K}}(h([\![\tau_1]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_m]), \ldots, h([\![\tau_m]\!]^{\mathbf{M}}[\mu_1, \mu_2, \ldots, \mu_m]))$$

Съгласно твърдение 3.5.4, това пък е еквивалентно на

$$\begin{aligned}\mathtt{p}^{\mathbf{K}}([\![\tau_1]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_m)], \ldots, [\![\tau_m]\!]^{\mathbf{K}}[h(\mu_1), h(\mu_2), \ldots, h(\mu_m)]) &\\ \longleftrightarrow \mathbf{K} \vDash \mathtt{p}(\tau_1, \ldots, \tau_m)[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]&\end{aligned}$$

Когато $\varphi = \psi_1 \,\&\, \psi_2$

$$\begin{aligned}&\mathbf{M} \vDash \psi_1 \,\&\, \psi_2[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \\ &\quad\longleftrightarrow \mathbf{M} \vDash \psi_1[\mu_1, \mu_2, \ldots, \mu_n] \text{ и } \mathbf{M} \vDash \psi_1[\mu_1, \mu_2, \ldots, \mu_n] \\ &\quad\longleftrightarrow \mathbf{K} \vDash \psi_1[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \text{ и } \mathbf{M} \vDash \psi_1[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \\ &\quad\longleftrightarrow \mathbf{K} \vDash \psi_1 \,\&\, \psi_2[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]\end{aligned}$$

Когато $\varphi = \psi_1 \vee \psi_2$ или $\varphi = \psi_1 \Rightarrow \psi_2$ се разсъждава аналогично.

Тъй като φ е безкванторна, то други възможности за φ няма. ∎

Току-що доказаното твърдение не е вярно за произволни формули. В теория на моделите се дефинират изображения, наречени *елементарни влагания*, които запазват верността на всички предикатни формули.

3.5.8. Да видим защо не може да докажем горното твърдение за произволни формули. Да разгледаме случая когато $\varphi = \forall \mathtt{x}_i\, \psi$. Тогава имаме:

$$\begin{aligned}&\mathbf{M} \vDash \forall \mathtt{x}_i\, \psi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \\ &\qquad\longleftrightarrow \text{за всяко } \nu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[\mu_1, \ldots, \mu_{i-1}, \nu, \mu_{i+1}, \ldots, \mu_n]\end{aligned}$$

Съгласно индукционното предположение последното е еквивалентно на

$$\text{за всяко } \nu \in |\mathbf{M}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), h(\nu), h(\mu_{i+1}), \ldots, h(\mu_n)] \quad (10)$$

От друга страна,

$$\mathbf{K} \vDash \forall \mathtt{x}_i\, \psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \longleftrightarrow$$
$$\longleftrightarrow \text{за всяко } \kappa \in |\mathbf{K}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), \kappa, h(\mu_{i+1}), \ldots, h(\mu_n)] \quad (11)$$

Като сравним (10) с (11) виждаме, че в единия случай твърдим, че нещо е вярно за всеки елемент от универсума на $\mathbf{K}$, а в другия случай — че същото нещо е вярно за всеки елемент от универсума на $\mathbf{K}$, който е от вида $h(\nu)$ за някое $\nu \in |\mathbf{M}|$.

Нека отбележим следното полезно за интуицията наблюдение. Когато формулата φ е безквантorна, тогава верността на $\mathbf{M} \vDash \varphi[\mu_1, \ldots, \mu_n]$ зависи единствено от свойствата на онези елементи на универсума на $\mathbf{M}$, които могат да се изразят явно посредством символите за константи, функционалните символи, както и елементите $\mu_1, \ldots, \mu_n$. Свойствата на останалите елементи на $\mathbf{M}$ не могат по никакъв начин да повлияят на верността на φ. С други думи, може да кажем, че безквантorните формули имат локален поглед към света. Кванторите пък позволяват на формулите да имат глобален поглед към целия свят. Например $\forall \mathtt{x}\, \psi$ означава, че нещо е вярно за всички елементи на универсума на $\mathbf{M}$ — дори онези, за които не разполагаме с никакъв начин да ги изразим явно.

Следващото твърдение показва, че е възможно да изкажем твърдение подобно на 3.5.7, в което вместо за обикновена вярност се говори за тъждествена вярност или изпълнимост, но вместо импликация ще имаме следване само в едната посока.

3.5.9. Твърдение. *Нека $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм. Тогава за всяка безквантorна формула φ:*

а) $\mathbf{K} \vDash \varphi \longrightarrow \mathbf{M} \vDash \varphi$*;*

б) Ако φ е изпълнима в $\mathbf{M}$, то φ е изпълнима в $\mathbf{K}$.

Доказателство. Нека формулата е $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$.

(а) $\mathbf{K} \vDash \varphi$ е вярно тогава и само тогава, когато за произволни елементи $\kappa_1, \kappa_2, \ldots, \kappa_n \in |\mathbf{K}|$ е вярно $\mathbf{K} \vDash \varphi[\kappa_1, \kappa_2, \ldots, \kappa_n]$. Но ако това е така, то в частност за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ ще бъде вярно

$\mathbf{K} \models \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$. От твърдение 3.5.7 следва, че за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ е вярно $\mathbf{M} \models \varphi[\mu_1, \mu_2, \ldots, \mu_n]$, следователно $\mathbf{M} \models \varphi$.

(б) $\mathbf{M} \models \varphi$ е вярно тогава и само тогава, когато съществуват такива $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$, че $\mathbf{M} \models \varphi[\mu_1, \mu_2, \ldots, \mu_n]$. От твърдение 3.5.7 следва, че за същите тези $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ е бъде вярно също и $\mathbf{K} \models \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$. Това означава, че съществуват такива $\kappa_1, \kappa_2, \ldots, \kappa_n \in |\mathbf{K}|$, че $\mathbf{K} \models \varphi[\kappa_1, \kappa_2, \ldots, \kappa_n]$, следователно φ е изпълнима в $\mathbf{K}$. ∎

3.5.10. Дефиниция. а) Ако $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм, което е биекция между универсумите на $\mathbf{M}$ и $\mathbf{K}$, казваме, че h е *изоморфизъм*.

б) Когато съществува изоморфизъм между структурите $\mathbf{M}$ и $\mathbf{K}$, казваме, че $\mathbf{M}$ е *изоморфна* на $\mathbf{K}$.

в) Когато $h\colon \mathbf{M} \to \mathbf{M}$ е изоморфизъм на една структура със същата структура, казваме, че h е *автоморфизъм*.

г) Изображението идентитет винаги е автоморфизъм. То се нарича *тривиален автоморфизъм*.

Забележка: Въпреки че има различни понятия за морфизми между структури (хомоморфизъм, силен хомоморфизъм, хомоморфно влагане, елементарно влагане и др.), оказва се, че понятието „изоморфизъм“ е абсолютно. Едно изображение е биективен силен хомоморфизъм, тогава и само тогава, когато е биективно хомоморфно влагане, тогава и само тогава, когато е биективно елементарно влагане.

3.5.11. Твърдение. *а) Всяка структура е изоморфна със себе си.*

б) Ако структурата $\mathbf{M}$ *е изоморфна с* $\mathbf{K}$*, то* $\mathbf{K}$ *е изоморфна с* $\mathbf{M}$*.*

в) Ако структурата $\mathbf{M}$ *е изоморфна с* $\mathbf{N}$ *и* $\mathbf{N}$ *е изоморфна с* $\mathbf{K}$*, то* $\mathbf{M}$ *е изоморфна с* $\mathbf{K}$*.*

Доказателство. Вж. твърдения 3.5.12, 3.5.13 и 3.5.14. ∎

3.5.12. Твърдение. *Нека* $\mathbf{M}$ *е структура, а* h *е изображението, което изобразява всеки елемент на универсума на* $\mathbf{M}$ *в същия елемент. Тогава* h *е изоморфизъм.*

Доказателство. Очевидно h е биекция. Освен това непосредствено може да се провери, че h удовлетворява изискванията на дефиницията за силен хомоморфизъм. ∎

3.5.13. Твърдение. *Ако $h\colon \mathbf{M} \to \mathbf{K}$ е изоморфизъм, то обратното изображение $h^{-1}\colon \mathbf{K} \to \mathbf{M}$ също е изоморфизъм.*

Доказателство. Да бъде h е биекция означава за произволни μ от универсума на $\mathbf{M}$ и κ от универсума на $\mathbf{K}$ да е вярно $h^{-1}(h(\mu)) = \mu$ и $h(h^{-1}(\kappa)) = \kappa$. Тъй като тези условия са симетрични спрямо μ и κ, то значи функцията h е обратна на h^{-1}, откъдето следва, че h^{-1} също е биекция.

Остава да докажем, че h^{-1} е силен хомоморфизъм. Това е така, защото

- за всеки символ за константа $\mathtt{c}$, $h^{-1}(\mathtt{c}^{\mathbf{K}}) = h^{-1}(h(\mathtt{c}^{\mathbf{M}})) = \mathtt{c}^{\mathbf{M}}$.
- за всеки n-местен функционален символ $\mathtt{f}$

$$\begin{aligned} h^{-1}(\mathtt{f}^{\mathbf{K}}(\kappa_1, \kappa_2, \ldots, \kappa_n)) = \\ &= h^{-1}(\mathtt{f}^{\mathbf{K}}(h(h^{-1}(\kappa_1)), h(h^{-1}(\kappa_2)), \ldots, h(h^{-1}(\kappa_n)))) \\ &= h^{-1}(h(\mathtt{f}^{\mathbf{K}}(h^{-1}(\kappa_1), h^{-1}(\kappa_2), \ldots, h^{-1}(\kappa_n)))) \\ &= \mathtt{f}^{\mathbf{K}}(h^{-1}(\kappa_1), h^{-1}(\kappa_2), \ldots, h^{-1}(\kappa_n)) \end{aligned}$$

- за всеки n-местен предикатен символ $\mathtt{p}$

$$\begin{aligned} \mathtt{p}^{\mathbf{K}}(\kappa_1, \kappa_2, \ldots, \kappa_n) &\longleftrightarrow \\ &\longleftrightarrow \mathtt{p}^{\mathbf{K}}(h(h^{-1}(\kappa_1)), h(h^{-1}(\kappa_2)), \ldots, h(h^{-1}(\kappa_n))) \\ &\longleftrightarrow \mathtt{p}^{\mathbf{K}}(h^{-1}(\kappa_1), h^{-1}(\kappa_2), \ldots, h^{-1}(\kappa_n)) \end{aligned}$$

■

3.5.14. Твърдение. *Нека $h_2\colon \mathbf{M} \to \mathbf{N}$ и $h_1\colon \mathbf{N} \to \mathbf{K}$ са силни хомоморфизми и h е композицията на h_1 и h_2, т.е. $h(\mu) = h_1(h_2(\mu))$ за всеки елемент μ на универсума на $\mathbf{M}$. Тогава $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм.*

Доказателство. Очевидно h изобразява елементите на $|\mathbf{M}|$ в елементи на $|\mathbf{K}|$. Освен това:

- За всеки символ за константа $h(\mathtt{c}^{\mathbf{M}}) = h_1(h_2(\mathtt{c}^{\mathbf{M}})) = h_1(\mathtt{c}^{\mathbf{N}}) = \mathtt{c}^{\mathbf{K}}$.
- За всеки n-местен функционален символ и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = h_1(h_2(\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)))$$

  Тъй като h_2 е силен хомоморфизъм, то последното е равно на

$$h_1(\mathtt{f}^{\mathbf{N}}(h_2(\mu_1), h_2(\mu_2), \ldots, h_2(\mu_n)))$$

Тъй като h_1 също е силен хомоморфизъм, това е равно на

$$\mathtt{f}^{\mathbf{K}}(h_1(h_2(\mu_1)), h_1(h_2(\mu_2)), \ldots, h_1(h_2(\mu_n))) = \mathtt{f}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

– Нека $\mathtt{p}$ е n-местен предикатен символ и $\mu_1, \mu_2, \ldots, \mu_n$ са елементи на универсума на $\mathbf{M}$. Тъй като h_2 е силен хомоморфизъм, то

$$\mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n) \longleftrightarrow \mathtt{p}^{\mathbf{N}}(h_2(\mu_1), h_2(\mu_2), \ldots, h_2(\mu_n))$$

Тъй като h_1 също е силен хомоморфизъм, това е еквивалентно на

$$\mathtt{p}^{\mathbf{K}}(h_1(h_2(\mu_1)), h_1(h_2(\mu_2)), \ldots, h_1(h_2(\mu_n))) \longleftrightarrow \\ \longleftrightarrow \mathtt{f}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

■

3.5.15. Твърдение. *Нека $h\colon \mathbf{M} \to \mathbf{K}$ е изоморфизъм. Тогава за произволни формула $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ и $\mu_1, \mu_2, \ldots, \mu_n$ от универсума на $\mathbf{M}$*

$$\mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \mathbf{K} \vDash \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Доказателство. С индукция по дължината на формулата φ. Случаите, когато φ е атомарна формула, $\bot$ или е получена с някоя съждителна операция, се разглеждат идентично както в доказателството на твърдение 3.5.7. Ще разгледаме само случаите с квантори.

Когато $\varphi = \forall \mathtt{x}_i\, \psi$

$$\mathbf{M} \vDash \forall \mathtt{x}_i\, \psi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \\ \longleftrightarrow \text{за всяко } \nu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[\mu_1, \ldots, \mu_{i-1}, \nu, \mu_{i+1}, \ldots, \mu_n]$$

Съгласно индукционното предположение последното е еквивалентно на

$$\text{за всяко } \nu \in |\mathbf{M}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), h(\nu), h(\mu_{i+1}), \ldots, h(\mu_n)]$$

Тъй като h е сюрекция, това е еквивалентно на

$$\text{за всяко } \kappa \in |\mathbf{K}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), \kappa, h(\mu_{i+1}), \ldots, h(\mu_n)]$$

т.е. на

$$\mathbf{K} \vDash \forall \mathtt{x}_i\, \psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Когато $\varphi = \exists \mathtt{x}_i\, \psi$

$$\mathbf{M} \vDash \exists \mathtt{x}_i\, \psi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \\ \longleftrightarrow \text{съществува такова } \nu \in |\mathbf{M}|, \text{ че } \mathbf{M} \vDash \psi[\mu_1, \ldots, \mu_{i-1}, \nu, \mu_{i+1}, \ldots, \mu_n]$$

Съгласно индукционното предположение последното е еквивалентно на

съществува такова $\nu \in |\mathbf{M}|$, че
$$\mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), h(\nu), h(\mu_{i+1}), \ldots, h(\mu_n)]$$

Тъй като h е сюрекция, това е еквивалентно на

съществува такова $\kappa \in |\mathbf{K}|$, че
$$\mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), \kappa, h(\mu_{i+1}), \ldots, h(\mu_n)]$$

т.е. на
$$\mathbf{K} \vDash \forall \mathtt{x}_i \, \psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

■

Забележка: От доказателството на твърдение 3.5.15 се вижда, че то е вярно не само за изоморфизми, но и за произволни сюрективни силни хомоморфизми.

Да напомним, че съгласно 3.3.11 б) когато пишем $\varphi(\mathtt{x})$ имаме предвид, че единствено $\mathtt{x}$ може да бъде свободна променлива на φ. С други думи $\mathtt{x}$ е единствената свободна променлива на φ или φ няма свободни променливи. Когато пишем $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, това означава, че всички свободни променливи на φ са измежду различните променливи $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

3.5.16. Дефиниция. Нека $\mathbf{M}$ е структура.

а) Подмножество X на универсума на $\mathbf{M}$ е *определимо* посредством формулата $\varphi(\mathtt{x})$, ако за всяко $\mu \in |\mathbf{M}|$
$$\mu \in X \longleftrightarrow \mathbf{M} \vDash \varphi[\mu]$$

б) Подмножество $X \subseteq |\mathbf{M}|^n$ е *определимо* посредством формулата $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, ако за произволни елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$
$$\langle \mu_1, \mu_2, \ldots, \mu_n \rangle \in X \longleftrightarrow \mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n]$$

3.5.17. Теорема за определимите множества. *Нека $\mathbf{M}$ е структура и $h\colon \mathbf{M} \to \mathbf{M}$ е произволен автоморфизъм.*

а) Ако подмножество X на универсума на $\mathbf{M}$ е определимо, то за всеки елемент μ на универсума на $\mathbf{M}$
$$\mu \in X \longleftrightarrow h(\mu) \in X$$

б) Ако подмножество $X \subseteq |\mathbf{M}|^n$ *е определимо, то за произволни елементи* $\mu_1, \mu_2, \ldots, \mu_n$ *на универсума на* $\mathbf{M}$

$$\langle \mu_1, \mu_2, \ldots, \mu_n \rangle \in X \longleftrightarrow \langle h(\mu_1), h(\mu_2), \ldots, h(\mu_n) \rangle \in X$$

Доказателство. **(а)** е частен случай на (б).

(б) Нека множеството X е определимо посредством формулата φ. Тъй като всеки автоморфизъм е изоморфизъм, съгласно твърдение 3.5.15

$$\begin{aligned} \langle \mu_1, \mu_2, \ldots, \mu_n \rangle \in X &\longleftrightarrow \mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n] \\ &\longleftrightarrow \mathbf{M} \vDash \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \\ &\longleftrightarrow \langle h(\mu_1), h(\mu_2), \ldots, h(\mu_n) \rangle \in X \end{aligned}$$

■

Накрая да отбележим, че ако добавим нов функционален или предикатен символ към сигнатурата и съответно към структурата, то това, което е било автоморфизъм в по-бедната структура, може да престане да бъде автоморфизъм в по-богатата структура.

Нека например структурата е с универсум реалните числа, единствен функционален символ е „+“, който се интерпретира като събиране и единствен предикатен символ е равенството. Тогава изображението

$$h(x) = -x$$

е автоморфизъм, защото:

1. h е биекция;
2. $h(x + y) = h(x) + h(y)$;
3. $x = y \longleftrightarrow h(x) = h(y)$.

Ако обаче обогатим структурата с нов функционален символ „.“, който се интерпретира като умножение, тогава h престава да бъде автоморфизъм. Всъщност, оказва се, че в обогатената структура единственият автоморфизъм е тривиалният автоморфизъм. Такива структури се наричат *твърди структури*.

Нещо аналогично се случва и във физиката. Например ако обърнем посоката на времето, фундаменталните закони на физиката остават в сила... Е, остават в сила, но с едно „незначително“ изключение за т. н. слабо взаимодействие. Можем да преведем на логически език това по следния начин — ако в сигнатурата не присъстват символи, изразяващи слабото взаимодействие, тогава обръщането на посоката на времето се оказва автоморфизъм, но ако обогатим структурата със слабото взаимодействие, тогава това престава да бъде автоморфизъм.

3.6. Интуиционистка логика

Класическа и конструктивна математика

При класическия начин за правене на математика съжденията, които формулираме, имат дескриптивен, т.е. описателен характер. Във всяко математическо твърдение се говори за свойствата, притежавани от математическите обекти. Самите математически обекти са статични и неизменни — по времето на древните гърци те са притежавали същите свойства, които притежават и днес. Математическите обекти сякаш живеят в свой идеален и неизменен свят. Математикът не се меси в този свят, не нарушава спокойствието му, а само като страничен наблюдател описва свойствата на математическите обекти. Ако изкажем съждението, че всяко реално число, повдигнато на квадрат, е неотрицателно, верността на това съждение няма нищо общо с това какво можем или не можем, какво искаме или не искаме, какво знаем или не знаем.

Конструктивният начин за правене на математика е алтернатива на класическия. При него също можем да формулираме дескриптивни съждения, но освен това обръщаме внимание и на това какво ние самите можем да правим с математическите обекти. Две числа не просто имат сбор, но могат да бъдат събирани, едно уравнение не просто има решение, но може да бъде решавано, производната на една функция не просто съществува, но може да бъде пресмятана и т. н. Например когато разработваме алгебрата конструктивно, ние не се задоволяваме с това да изкажем твърдение, според което полиномите с комплексни коефициенти имат комплексен корен, а започваме да изследваме въпроса дали има начин, с който да можем да намерим корена. С други думи, когато правим математиката конструктивно, в математическите твърдения се говори не само за идеални математически обекти, но и за нас самите — какво можем или не можем да правим с математическите обекти.

Забележка: Има различни философски теории за математиката. Въз основа на тези философии много от конструктивистите не биха се съгласили с обясненията, които ще дадем за това какво представлява и как се прави конструктивната математика. Тук обаче няма да обръщаме никакво внимание на тези противоречащи си една на друга философии, нито на това кое според тях е „правилно“ и кое „неправилно“. В частност няма да се интересуваме от това дали някоя математическа теория — конструктивна или не — противоречи на едни или други философски принципи. Математическа значимост, а значи и значимост от

гледна точка на практическите приложения, имат единствено следните три неща:

– Кои съждения могат да бъдат формулирани, използвайки езика на дадена математическа теория?

– Каква част от съжденията теорията може да докаже?

– Какви допускания теорията приема без доказателство?

Ще отбележим все пак едно по-съществено несъответствие между тукашните обяснения и едно от най-важните конструктивни направления в математиката — *интуиционизма*. Според обясненията, които даваме тук, също както и в класическата математика, ако допуснем, че даден обект не съществува и стигнем до противоречие, това означава, че този обект все пак съществува, макар и не винаги да сме състояние да го намерим. Това означава, че в нашите разсъждения предполагаме съществуването на някакъв „свят“, в който съществуват математическите обекти. Ако допуснем, че в този свят даден обект не съществува и стигнем до противоречие, значи въпросният обект съществува. От друга страна, според математическия интуиционизъм не е правомерно да предполагаме съществуването на математически свят, в който има математически обекти, за които не можем дори въобразаемо да допуснем, че могат да бъдат конструирани. Можем да считаме, че съществува само това, което наистина можем конструираме и можем да считаме за вярно само това, което можем да докажем, че е вярно. Ако допуснем, че даден обект не съществува, ние всъщност допускаме, че не сме в състояние да конструираме въпросния обект. Ако това ни доведе до противоречие, това означава, че никога няма да можем да стигнем до извода, че не можем да конструираме дадения обект. С други думи, ако допуснем, че даден обект не съществува и стигнем до противоречие, това означава само, че винаги ще съществува надеждата, че този обект съществува, но само това — няма никакви гаранции, че обектът наистина съществува.

Интуиционистка логика

Това че в конструктивната математика можем да формулираме и доказваме съждения, които не могат дори да се формулират на езика на класическата математика, може да ни наведе на мисълта, че конструктивната математика се нуждае от логика с нови логически операции, при която твърденията трябва да се доказват по принципно нов начин. За щастие, оказва се, че това съвсем не е така. Например всички неща, които са доказани в този математически текст до момента, са

доказани конструктивно и за да направим това дори не ни се наложи да предупредим предварително читателя. Използваните тук доказателства на различните твърдения са конструктивни, но в същото време те могат да се четат и от хора, които никога не са чували за конструктивната математика, нито пък знаят какво означава едно доказателство да бъде конструктивно.

Как става възможно това? Начинът е следният — ще използваме обичайните логически операции, но ще ги тълкуваме по нов начин. Това между другото означава, че конструктивната математика се прави донякъде на принципа „говорим едно, ама разбираме друго“.

Когато един конструктивист изкаже твърдение от вида

„съществува такова x, че ...“,

той всъщност има предвид

„може да се намери такова x, че ...“.

Когато пък конструктивистът изкаже твърдение от вида

„A или B“,

той всъщност има предвид

„може да се установи дали A, или B“.

За останалите логически операции може да считаме, че те в конструктивната математика се тълкуват по същия начин, както и в класическата.

Дали това, че тълкуваме някои от логическите операции по друг начин, означава, че в конструктивната математика трябва да се използват по-различни доказателства, в сравнение с класическата? Удивително е, че отговорът на този въпрос е отрицателен — логиката, която се използва в конструктивната математика, наречена *интуиционистка логика*, не се нуждае от по-различен вид доказателства в сравнение с класическата логика.* Достатъчно ще бъде да спазваме две прости правила, и това ще ни гарантира, че доказателството е конструктивно.

Първото правило е следното: не трябва да използаме метода „допускане на противното“. Ако допуснем A и стигнем до противоречие, това ще ни гарантира, че A не е вярно, но ако допуснем, че A не е вярно и стигнем до противоречие, ние ще докажем не съждението A, а

*Всъщност ако интуиционистката логика се нуждаеше от по-различен вид доказателства, то не би имало особен смисъл да използваме обичайните класически логически операции, пък да ги тълкуваме по друг начин. Вместо „съществува x“ можеше да казваме „може да се намери x“ и вместо „A или B“ можеше да казваме „може да установим дали A, или B“.

неговото двойно отрицание. Ще видим, че когато разсъждаваме конструктивно, двойното отрицание на едно съждение не винаги може да се счита еквивалентно на самото съждение.

Второто правило за правене на конструктивни доказателства е следното: имаме право да разглеждаме случаи за това дали едно съждение е вярно, или не, само тогава, когато разполагаме с метод, посредством който можем да проверим дали съждението е вярно, или не.

По-нататък в този раздел ще видим с конкретни примери как спазването на тези две прости правила наистина ще ни позволи да доказваме твърденията по такъв начин, че доказателствата да са верни без значение дали тълкуваме формулировките на съждения както в конструктивната математика, или както в класическата.

Двойно отрицание

Нека видим защо в интуиционистката логика двойното отрицание не се унищожава. Нека A е следното твърдение:

> „Съществува цифра, която се среща безброй много пъти в десетичното представяне на π“.

Тъй като има само десет различни цифри — 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9 — то няма как всяка от тях да се среща само краен брой пъти в десетичното представяне на числото π. Следователно твърдението A е вярно.

Ако обаче изтълкуваме формулировката на това твърдение конструктивно, тогава смисълът му ще стане следният:

> „Може да се намери цифра, която се среща безброй много пъти в десетичното представяне на π“.

Засега за нито една от десетте цифри не знаем със сигурност дали се среща безброй много пъти в десетичното представане на π и затова ако тълкуваме формулировката на твърдение A конструктивно, тогава това твърдение не може да се счита за доказано. Най-вероятно всяка цифра среща безброй много пъти, но засега това не е нищо повече от предположение.

Да допуснем, че твърдението A е невярно, означава да допуснем, че не съществува цифра, която се среща безброй много пъти в десетичното представяне на π. От тук не е трудно да се стигне до противоречие, следователно това допускане не е вярно. Щом като допускането, че твърдението A не е вярно, води до противоречие, значи е вярно двойното отрицание на A.

И така, доказахме двойното отрицание на A, макар че засега самото твърдение A остава недоказано.

При интуиционистката логика всяко математическо твърдение носи определена информация. Когато информацията представлява някой конкретен математически обект или метод за преобразуване на математически обекти, казваме, че тази информация е конструктивна. Например информацията за конкретна цифра, която се среща безброй пъти в π е конструктивна. Когато пък информацията не ни дава конкретни математически обекти или методи, а само сведения за свойствата на математическите обекти, казваме, че тази информация е дескриптивна. Например информацията, която казва, че има цифра, която се среща безброй пъти в π, но не ни казва коя точно е тази цифра, е дескриптивна.

Какъв е интуитивният смисъл на двойното отрицание? Оказва се, че можем да използваме двойно отрицание, за да премахнем конструктивната информация от едно съждение. Например твърдението A ни дава конкретна цифра, която се среща безброй пъти в десетичното представяне на π. Тъй като засега не знаем коя е тази цифра, то твърдението A все още няма конструктивно доказателство. Отрицанието на A казва, че няма цифра, която се среща безброй пъти, и значи двойното отрицание на A казва, че не е възможно да няма цифра, която се среща безброй пъти. Е, щом не е възможно да няма цифра, която се среща безброй пъти в десетичното представяне на π, значи такава цифра съществува. Твърдението „не не A“ обаче не ни дава информация коя е тази цифра — видяхме, че ние можем да докажем това твърдение дори и без да знаем коя точно цифра се среща безброй пъти.

3.6.1. Факт. *Съждението A е дескриптивно, т.е. не носи конструктивна информация, тогава и само тогава, когато A е еквивалентно на „не не A“.*

Да отбележим, че няма смисъл да правим тройно отрицание на едно съждение, защото тройното отрицание е еквивалентно на единичното. Това е така, защото единичното отрицание на едно съждение просто казва, че нещо е невъзможно или невярно, и не ни носи никаква конструктивна информация. Единственото нещо, което прави двойното отрицание, пък е това да премахне конструктивната информация от едно съждение. Затова ако приложим двойно отрицание към съждение, което не съдържа конструктивна информация, например към единично отрицание, резултатът ще бъде еквивалентно на него съждение.

Твърдение. *За произволно съждение A*

а) от A следва „не не A“;

б) от „не A“ следва „не не не A“;

в) от „не не не A“ следва „не A“;

г) „не не не A“ е еквивалентно на „не A“.

Доказателство. **(а)** Да допуснем, че A е вярно. Трябва да докажем „не не A“. За да докажем това, да допуснем, че „не A“. От това веднага получаваме противоречие (имаме едновременно A и не A) и значи допускането „не A“ е невъзможно. Значи е вярно „не не A“.

(б) се получава от (а) като сложим „не A“ на мястото на A.

(в) Да допуснем, че „не не не A“. Трябва да докажем, че „не A“. За да докажем това, да допуснем, че A е вярно. От тук и от (а) получаваме „не не A“, а това ни дава противоречие (имаме „не не A“ и отрицанието на „не не A“). Противоречието се дължи на допускането, че A е вярно и значи A не е вярно.

(г) следва от (б) и (в). ∎

***Забележка*:** От тук нататък, когато искаме да изкажем съждение с двойно отрицание, ще използваме изрази от вида „неможе да не е вярно, че ...“. Например следното твърдение е вярно конструктивно: „не може да не съществува цифра, която се среща безброй много пъти в десетичното представяне на числото π“.

Превод на класическата логика в интуиционистката

Благодарение на двойното отрицание, всичко, което можем да правим в класическата математика, може да бъде направено и в конструктивната. Вече споменахме, че има две логически операции, които в интуиционистката логика се тълкуват по-различно, отколкото в класическата — кванторът за съществуване и дизюнкцията. Конструктивното съществуване ни казва кой точно е съществуващият обект, докато класическото съществуване ни казва, че обектът съществува по принцип, без да ни дава метод как да намерим този обект. Конструктивната дизюнкция „A или B“ ни казва дали A е вярно, или B, докато при класическата дизюнкция може и да не знаем кое точно от тези две съждения е вярно. Също така видяхме, че можем да използваме двойно отрицание, за да премахваме конструктивната информация от едно съждение. Това означава, че можем да използваме двойно отрицание,

за да преведем всяко съждение, използващо класическата логика, на езика на интуиционистката логика.

3.6.2. Факт. *Ако вземем съждение, изказано използвайки класическата логика, и пред всеки квантор за съществуване, както и пред всяка дизюнкция, сложим двойно отрицание, ще получим равносилно на него съждение, изказано използвайки интуиционистката логика.*

Този превод на класическата логика в интуиционистката се нарича *отрицателен превод*.

Конструктивната математика включва в себе си класическата математика, но я разширява по съществен и нетривиален начин. Така например Аритметиката на Пеано представлява формална теория, в която могат да се формулират съждения за естествени числа. Конструктивен аналог на Аритметиката на Пеано е Аритметиката на Хейтинг. Всяко съждение, което може да се формулира на езика на Аритметиката на Пеано, може да се преведе и на езика на Аритметиката на Хейтинг и всяко съждение, което може да се докаже от първата теория, може да се докаже и от втората. На езика на Аритметиката на Хейтинг обаче могат да се формулират и много съждения, които не могат да се изкажат на езика на Аритметиката на Пеано.*

Аналогично е положението и с Теорията на Цермело – Френкел (ZF), която представлява формална теория, в която могат да се формулират съждения за множества и обикновено се използва като основа на съвременната класическа математика. Конструктивен аналог на ZF е Интуиционистката теория на Цермело – Френкел (IZF). Всяко съждение, което може да се формулира на езика на ZF, може да се преведе и на езика на IZF и всяко съждение, което може да се докаже от първата теория, може да се докаже и от втората. На езика на IZF обаче могат да се формулират и много съждения, които не могат да се изкажат на езика на ZF.

Забележка: Исторически конструктивната математика възникнала във връзка със желанието да се даде по-обоснован фундамент на математиката. Считало се, че класическата математика е необоснована и заплашена от вътрешни противоречия, докато съждения в конструктивната математика имат ясен смисъл и затова е много невероятно в

*Едно такова съждение е *тезисът на Чърч*. На езика на Аритметиката на Хейтинг тезисът на Чърч може да се формулира така:

$$\forall \mathtt{x}\, \exists \mathtt{y}\, \varphi[\mathtt{x}, \mathtt{y}] \Rightarrow \exists \mathtt{e}\, \forall \mathtt{x}\, (!\{\mathtt{e}\}(\mathtt{x}) \,\&\, \varphi[\mathtt{x}, \{\mathtt{e}\}(\mathtt{x})])$$

нея да се открие противоречие. Реалното сравнение на теоретикодоказателствената сила на различни класически и конструктивни теории показва, че това мнение е неправилно. Например ако някога се открие противоречие в теорията ZF, то и теорията IZF ще се окаже противоречива. Въпросът, който има реално значение за основите на математиката, е не това дали да правим математиката конструктивно, или не, а това дали да позволяваме т. н. *непредикативни дефиниции*. Мнението, че конструктивната математика е „по-обоснованият“ начин за правене на математика, се дължи на това, че теорията ZF е непредикативна, докато почти всички математици-конструктивисти предпочитат да правят математиката предикативно. Вместо IZF те използват теорията CZF (която е предикативен аналог на IZF), теория на типовете или дори език без множества.

Изоморфизъм на Къри – Хауърд

Приложенията на интуиционистката логика далеч не се изчерпват само с това, че тя ни позволява да разработваме математиката конструктивно. Оказва се, че всяко съждение, изказано посредством интуиционистката логика, може да се интерпретира като тип данни, а всяко интуиционистко доказателство ни дава обект, притежаващ съответния тип данни. Вярно е и обратното — типовете данни в езиците за програмиране може да се интерпретират като интуиционистки съждения, а обектите, дефинирани в компютърните програми — като интуиционистки доказателства. Тази връзка между логика и програмиране се нарича *изоморфизъм на Къри – Хауърд*.

За да си обясним как е възможно съжденията да бъдат типове данни, а програмните обекти — доказателства, да разгледаме един пример. Нека A е съждението

„Можем да намерим мегапираня“,

а B е съждението

„Можем да намерим хипопозавър“.

Конструктивната информация, която се съдържа в съждението A, е метод за намиране на мегапираня, а конструктивната информация, която се съдържа в съждението B, е метод за намиране на хипопозавър. Да докажем съждението A означава да посочим метод за намиране на мегапираня, а да докажем B означава да посочим метод за намиране на хипопозавър. На съждението A съответства типът данни „метод за откриване на мегапираня“, а на съждението B — „метод за откриване на хипопозавър“. Всеки обект, чийто тип е „метод за откриване на ме-

гапираня“, може да се счита за доказателство на съждението A и всеки обект, чийто тип е „метод за откриване на хипопозавър“, може да се счита за доказателство на съждението B.

Нека видим как изглежда това съответствие между съждения и типове и между доказателства и програмни обекти при различните логически операции.

Конюнкция

Съждението „A и B“ казва, че A и B са верни, т.е. разполагаме с метод за откриване на мегапираня и метод за откриване на хипопозавър. Това означава, че информацията, която се съдръжа в съждението „A и B“ е комбинация от информацията, която се съдържа в A, и информацията, която се съдържа в B.

Следователно на съждението „A и B“ съответства следният тип данни: „наредена двойка от метод за откриване на мегапираня и метод за откриване на хипопозавър“. На хаскел този тип се обозначава така:

`(`A`,`B`)`

В математиката доказваме конюнкцията по следния начин:

– Доказваме A.

– Доказваме B.

– От тук заключаваме, че е вярно „A и B“.

Ако x е обект от тип A, а y е обект от тип B, тогава наредената двойка от x и y на хаскел се обозначава така:

`(`x`,`y`)`

Следователно, ако си мислим x като доказателство на A и y — като доказателство на B, то `(`x`,`y`)` ще бъде доказателство на „A и B“

Обратно, ако вече сме доказали „A и B“, от тук директно можем да получим като следствия A и B. Също и на хаскел ако вече разполагаме с обект z от тип `(`A`,`B`)`, тогава

`fst` z

ще ни даде първия елемент на z, а

`snd` z

ще ни даде втория. Следователно ако z е доказателство на „A и B“, то „`fst` z“ ще бъде доказателство на A, а „`snd` z“ ще бъде доказателство на B.

Дизюнкция

Когато тълкуваме дизюнкцията конструктивно, съждението „A или B“ има следния смисъл:

> „Може да се установи дали можем да намерим мегапираня,
> или можем да намерим хипопозавър.“

Следователно съждението „A или B“ ни дава метод за откриване на мегапираня или метод за откриване на хипопозавър.

На съждението „A и B“ съответства тип данни, който в известен смисъл представлява обединение на типовете A и B. Всеки обект от тип „A или B“ ни дава обект от тип A или обект от тип B. На хаскел този тип се обозначава така:

`Either` A B

В математиката доказваме дизюнкцията по следните два начина:

– Доказваме A.

– От тук заключаваме, че е вярно „A или B“.

Втори начин:

– Доказваме B.

– От тук заключаваме, че е вярно „A или B“.

Също и на хаскел ако x е обект от тип A, тогава

`Left(`x`)`

е обект от тип „`Either` A W“, където на мястото на W може да стои произволен тип. Значи ако си мислим x като доказателство на A, тогава „`Left(`x`)`“ представлява доказателство на „A или B“.

Аналогично, ако y е обект от тип B, тогава

`Right(`y`)`

е обект от тип „`Either` W B“, където на мястото на W може да стои произволен тип. Ако си мислим y като доказателство на B, тогава „`Right(`y`)`“ представлява доказателство на „A или B“.

Обратно, ако вече сме доказали „A или B“, начинът да използваме това съждение е да разглеждаме случаи:

– Първи случай — вярно е A. Използвайки A, доказваме някакво съждение C.

– Втори случай — вярно е B. Използвайки B, доказваме някакво съждение C.

– Тъй като и в двата случая получихме C, заключаваме, че C е вярно.

Аналогично на това, и на хаскел ако вече разполагаме с обект z от тип `Either` A B, един начин да го използваме е следната конструкция:

```
u = case z of
   Left(x) →
      използвайки x намираме обект от тип C
   Right(y) →
      използвайки y намираме обект от тип C
```

Импликация

Съждението „ако A, то B“ има следния смисъл:

> „Ако можем да намерим мегапираня, то ще можем да намерим хипопозавър.“

Следователно съждението „ако A, то B“ ни дава метод, посредством който ако ни бъде даден метод за откриване на мегапираня, ще можем да получим метод за откриване на хипопозавър.

На съждението „ако A, то B“ съответства типът данни „функция, чийто аргумент е от тип A (т.е. метод за откриване на мегапираня), а стойността — от тип B (т.е. метод за откриване на хипопозавър)“. На хаскел този тип се обозначава така:

A `->` B

В математиката доказваме импликацията по следния начин:

– Допускаме, че A е вярно.

– Започваме да правим разсъждения, използващи A.

– В края на тези разсъждения доказваме B

– От тук заключаваме, че от A следва B.

На доказателство от този вид съответства приблизително следният код на хаскел:

```
\x →
   let ...
       ...използвайки x дефинираме разни обекти
       ...
   in
      израз от тип B
```

Този код представлява анонимна функция с аргумент `x` от тип A, която връща стойност от тип B. За да можем да дадем на хаскел по-конкретен

пример, да разгледаме съждението „ако A и B, то B и A“. Това съждение е очевидно вярно. На хаскел на него съответства функция, която взема като аргумент наредена двойка `(a,b)` и връща като стойност наредената двойка `(b,a)`:

```
\x →                    -- x е аргументът на функцията
   let a = fst(x)       -- приемаме, че x е
       b = snd(x)       -- наредената двойка (a,b)
   in
      (b,a)             -- връщаме като стойност (b,a)
```

Ако вече сме доказали съжденията A и „ако A, то B“, то от тях ще получим като следствие и съждението B.

Аналогично на това, и на хаскел ако x е обект от тип A и f е функция от тип A`->`B, то $f(x)$ ще бъде обект от тип B.

Отрицание

Отрицанието в интуиционистката логика има две особености, които ще приемем без да ги обосноваваме.

Първата особеност на отрицанието е следната: приемаме, че ако дадено съждение не е вярно, то от него следва всяко съждение. Тази особеност на интуиционистката логика е присъща и на класическата логика, а значи и на цялата математика.

Втората особеност на отрицанието е следната. Да си представим на ум, че по някакъв начин сме се сдобили с мегапираня, и да допуснем, че от тук можем да стигнем до противоречие. С други думи, допускането, че по някакъв начин сме намерили мегапираня, води до противоречие. Ще можем ли да стигнем до противоречие ако допуснем, че съществува мегапираня без да допускаме, че сме намерили тази мегапираня? Не е ясно, обаче интуиционистката логика приема, че отговорът на този въпрос е положителен.

Смисълът на съждението „не A“ е следният:

„Не съществува мегапираня.“

Начинът, по който доказваме такова съждение е следният:

– Допускаме, че A е вярно.

– Започваме да правим разсъждения, използващи A.

– В края на тези разсъждения стигаме до противоречие

– От тук заключаваме, че A не е вярно.

Сред типовете на хаскел няма непосредствено представяне на отрицанието. Също както в този курс приехме, че всяка формула от вида $\neg\varphi$ е съкратен запис на $\varphi \Rightarrow \bot$, така и при хаскел, за да представим отрицанието, използваме свеждане към противоречие. Това е коректно, защото съжденията „A не е вярно“ и „от A следва противоречие“ са еквивалентни и значи може да считаме, че на съждението „не A“ съответства типът A `->` `Impossible`. Тук `Impossible` е празният тип, т.е. тип който няма нито една стойност и съответства на нашата формула $\bot$. Празният тип не е вграден, но в хаскел 2010 той може да се дефинира така:

```
data Impossible
```

Квантор за всеобщност

Да разгледаме съждението

> „За всеки зъб на мегапираня може да се намери мегапираня, която го е притежавала.“

Това съждение ни дава метод, посредством който ако ни бъде даден зъб от мегапираня, ще можем да получим метод за намиране на мегапиранята, която е притежавала този зъб. На това съответства типът данни „функция, чийто аргумент е обект x от тип „мегапирански зъб“, а стойността от тип „мегапираня, притежател на x“.

На пръв поглед, разгледан като тип данни, кванторът за всеобщност се интерпретира по същия начин, както и импликацията — като функция. Има обаче и една съществена разлика. При импликацията типовете на аргумента и на стойността са фиксирани и отнапред известни. При квантора за всеобщност аргументът също е има фиксиран тип, но типът на върнатата стойност зависи от аргумента, защото в израза „мегапираня, притежател на x“ се споменава x. Такива типове се наричат *зависими* (dependent types).

Един друг пример за зависим тип ни дава функцията, която получава като аргумент някакво естествено число n и връща като стойност n-мерен нулев вектор. Аргументът на тази функция е от тип „естествено число“, а връщаната стойност — „n-мерен вектор“ е зависим тип, защото зависи от n.

В хаскел няма вградена поддръжка за зависими типове,[*] но често няма проблем да пишем програми и да си мислим, че те използват

[*] Има начин поведението им донякъде да се имитира, вж. [12].

зависими типове — просто компилаторът ще пресметне някакви по-общи типове. Например разгледаната по-горе функция, която връща n-мерен нулев вектор, ще връща стойност от тип „вектор“ без да се уточнява, че векторът е n-мерен.

В математиката доказваме съждение от вида „за всяко x е вярно $C(x)$“ по следния начин:

– Избираме произволен обект x.

– Започваме да правим разсъждения за x.

– В края на тези разсъждения доказваме $C(x)$.

– От тук заключаваме, че за всяко x е вярно $C(x)$.

На доказателство от този вид съответства приблизително следният код на хаскел:

```
\x →
   let ...
       ... използвайки x дефинираме разни обекти
       ...
   in
      израз от тип C(x)
```

Този код представлява анонимна функция с аргумент `x`, която връща стойност от тип $C(x)$.

Ако вече сме доказали съждението „за всяко x е вярно $C(x)$“, то за всеки конкретен обект x ще можем да получим като следствие $C(x)$.

Аналогично на това, и на хаскел ако f е функция която по аргумент x ни връща обект от тип $C(x)$, то $f(x)$ ще бъде обект от тип $C(x)$.

Квантор за съществуване

Да разгледаме съждението

> „Съществува мегапираня, която е загубила зъб в река Ориноко.“

От това съждение можем да получим следната информация:

– някоя мегапираня;

– зъб, загубен от тази мегапираня в река Ориноко.

На това съответства типът данни наредена двойка, чийто пръв елемент x е от тип „мегапираня“, а вторият — от тип „зъб, загубен от x в река Ориноко“.

На пръв поглед, разгледан като тип данни, кванторът за съществуване се интерпретира по същия начин както конюнкцията — като

наредена двойка. Има обаче и една съществена разлика — типът на втория елемент на тази наредена двойка зависи от x и значи е зависим тип.

Няколко примера на хаскел

В следващите няколко примера най-напред ще дадем програмен код на хаскел, а след това ще коментираме как той може да се интерпретира като математическо доказателство.

Когато на хаскел дефинираме някакъв обект A, с инструкцията `:t` A можем да получим неговия тип. Навсякъде в дадените примери `Prelude` е командният показалец на интерпретатора — текстът след него се въвежда от нас. Текстът пък, който не е предшестван от команден показалец, се отпечатва от интерпретатора. Например в следния пример

```
Prelude> let x = 'q'
Prelude> :t x
x :: Char
```

на първия ред дефинираме променлива `x` със стойност `'q'`. На втория ред питаме интерпретатора какъв е типът на `x`. На третия ред интерпретаторът отговаря, че типът на `x` е `Char`. За да може да се въвеждат по-долните примери директно от командния ред на интерпретатора (вместо да се записват във файл), трябва да се използва командата

```
Prelude> :set +m
```

Първият пример, с който ще илюстрираме съответствието между програми и доказателства, е следният:

```
Prelude> let composition(f,g) = h
Prelude|         where h(x) = g(f(x))
Prelude|
Prelude> :t composition
composition :: (t2 -> t1, t1 -> t) -> t2 -> t
```

От програмна гледна точка тук дефинираме функцията `composition`, която получава като аргумент наредена двойка от две функции `f` и `g` и връща като стойност тяхната композиция `h`. Ако превърнем типа на функцията `composition` в логическа формула ще получим формулата

$$(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi) \Rightarrow (\varphi \Rightarrow \chi)$$

в която φ, ψ и χ са формулите, отговарящи съответно на автоматичните типове `t2`, `t1` и `t`. Да забележим, че тази формула е тавтология. Функцията `f`, която `composition` получава като аргумент, е от тип `t2->t1`,

т.е. $\varphi \Rightarrow \psi$, а функцията `g` — от тип `t1->t`, т.е. $\psi \Rightarrow \chi$. Стойността `h`, която `composition` трябва да върне е от тип $\varphi \Rightarrow \chi$. За да получим `h`, да допуснем, че притежаваме обект `x` от тип φ. Ако към `x` приложим `f`, ще получим обект от тип ψ. Ако към този обект приложим `g`, ще получим обект от тип χ. По този начин получаваме функция, която получава аргумент от тип φ и връща стойност от тип χ и значи типът на тази функция е $\varphi \Rightarrow \chi$.

Това е „програмистката“ интерпретация на случващото се във функцията `composition`. Да видим сега какво доказателство съответства на тази функция. Тъй като аргументът ѝ е от тип $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi)$, то доказателството започва с изречението „Да допуснем, че е вярно $(\varphi \Rightarrow \psi)\,\&\,(\psi \Rightarrow \chi)$“. В дефиницията на функцията `composition`, аргументът ѝ е наредената двойка `(f,g)`. Това е все едно в доказателството за удобство да означим $(\varphi \Rightarrow \psi)$ с `f` и $(\psi \Rightarrow \chi)$ с `g`. Тъй като `composition` връща като стойност функцията `h`, то доказателството продължава според дефиницията на функцията `h`. Аргументът `x` на `h` е от тип φ, и значи доказателството продължава с изречението „Да допуснем, че е вярно φ“. След това h връща `g(f(x))`. На това отговаря следното изречение „От φ и `f` получаваме ψ, а от тук и `g` получаваме χ“.

И така, на дадената по-горе програма отговаря следното доказателство:

Да допуснем, че $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi)$. Тогава

$$\varphi \Rightarrow \psi \qquad (\texttt{f})$$

и

$$\psi \Rightarrow \chi \qquad (\texttt{g})$$

Да допуснем, че е вярно φ. От φ и `f` получаваме ψ, а от ψ и `g` получаваме χ.

Допуснахме φ и доказахме χ. Значи е вярно

$$\varphi \Rightarrow \chi \qquad (\texttt{h})$$

Допуснахме $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi)$ и доказахме $\varphi \Rightarrow \chi$, значи е вярно

$$(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi) \Rightarrow \varphi \Rightarrow \chi \qquad (\texttt{composition})$$

Да разгледаме друг пример.

```
Prelude> let dis(x) = Left x
Prelude> :t dis
dis :: a -> Either a b
```

От програмна гледна точка тук дефинираме функцията `dis`, която получава обект от произволен тип `a` и връща като стойност обекта `Left(x)` от тип `Either a b`. Да си припомним, че на типа `Either a b` съответства формулата $\varphi \vee \psi$, където φ е формулата, съответстваща на типа `a`, а ψ е формулата, съответстваща на типа `b`. Следователно формулата, съответстваща на типа на функцията `dis` е

$$\varphi \Rightarrow \varphi \vee \psi$$

Да забележим, че тази формула е тавтология.

Извикването на функцията `dis` се свежда до извикването на конструктора `Left`. Типът `Either a b` има още един конструктор — `Right`, с чиято помощ можем да докажем и формулата $\psi \Rightarrow \varphi \vee \psi$.

Разгледаният пример с дизюнкция дава тривиално от математическа гледна точка доказателство. Ето един по-интересен пример, също използващ дизюнкция:

```
Prelude> let cases(f,g) = h
Prelude|         where
Prelude|             h(x) = case x of
Prelude|                    Left(y) -> f(y)
Prelude|                    Right(z) -> g(z)
Prelude|
Prelude> :t cases
cases :: (t1 -> t, t2 -> t) -> Either t1 t2 -> t
```

Преведен като формула, типът на функцията `cases` е

$$(\varphi \Rightarrow \chi)\ \&\ (\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi)$$

където φ, ψ и χ са формулите, съответстващи на автоматичните типове `t1`, `t2` и `t`. Функцията `h`, която функцията `cases` връща като стойност, има за аргумент обект `x` от тип `Either t1 t2`, т.е. $\varphi \vee \psi$. Тази функция разглежда два случая в зависимост от това дали `x` има вида `Left(y)`, където `y` е от тип φ, или `Right(z)`, където `y` е от тип ψ. В първия случай `h` връща като стойност `f(y)`, а във втория — `g(z)`.

Да видим как да интерпретираме това като математическо доказателство на формулата $(\varphi \Rightarrow \chi)\ \&\ (\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi)$. Щом като аргументът на `cases` е от тип $(\varphi \Rightarrow \chi)\ \&\ (\psi \Rightarrow \chi)$, то значи доказателството започва с думите „Да допуснем, че $(\varphi \Rightarrow \chi)\ \&\ (\psi \Rightarrow \chi)$“. Аргументът на `h` има вида `(f,g)`, което е все едно в математическото доказателство за краткост да означим $\varphi \Rightarrow \chi$ с `f` и $\psi \Rightarrow \chi$ с `g`. Функцията `cases` връща като стойност функцията h, така че доказателството продължава според дефиницията на `h`. Функцията `h` има аргумент от тип $\varphi \vee \psi$, което

значи, че доказателството продължава с изречението „Да допуснем, че е вярно $\varphi \vee \psi$“. След това в `h` се разглеждат случаи според вида на аргумента на `h`. В математическото доказателство това отговаря на това да разгледаме случаи в зависимост от това дали е вярно φ или ψ. При първия случай пресмятаме `f(y)`, където `y` е от тип φ, а `f` е от тип $\varphi \Rightarrow \chi$. В математическото доказателство това отговаря на изречението: „Щом като φ е вярно, а според `f` от φ следва χ, то значи χ е вярно.“ Във втория случай пресмятаме `g(z)`, където `z` е от тип ψ, а `g` е от тип $\psi \Rightarrow \chi$. В математическото доказателство това отговаря на изречението: „Щом като ψ е вярно, а според `g` от ψ следва χ, то значи и χ е вярно.“

И така, на дадената по-горе програма отговаря следното доказателство:

Да допуснем, че $(\varphi \Rightarrow \chi)\ \&\ (\psi \Rightarrow \chi)$ Тогава

$$\varphi \Rightarrow \chi \qquad (\texttt{f})$$

и

$$\psi \Rightarrow \chi \qquad (\texttt{g})$$

Да допуснем, че

$$\varphi \vee \psi \qquad (\texttt{x})$$

и да разгледаме случаи в зависимост от това дали е вярно φ или ψ.

В първия случай, когато е вярно φ, от `f` получаваме χ.

Във втория случай, когато е вярно ψ, от `g` отново получаваме χ.

И в двата случая получихме χ.

Допуснахме $\varphi \vee \psi$ и доказахме χ, значи е вярно

$$\varphi \vee \psi \Rightarrow \chi \qquad (\texttt{h})$$

Допуснахме $(\varphi \Rightarrow \chi)\ \&\ (\psi \Rightarrow \chi)$ и доказахме $\varphi \vee \psi \Rightarrow \chi$, значи е вярно

$$(\varphi \Rightarrow \chi)\ \&\ (\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi) \qquad (\texttt{cases})$$

3.7. Нормални форми

В много дялове на математиката се използват т. н. нормални форми. Когато всеки обект от някой тип може да бъде представен посредством израз, имащ определен вид, казваме, че този израз е нормалната форма на обекта. Когато искаме да дефинираме в компютърна програма тип, чиито обекти притежават нормална форма, не е нужно да се грижим за компютърно представяне на изрази, които не са в нормална форма.

Ето няколко примера.

– От дискретната математика знаем, че булевите функции могат да се представят в конюнктивна нормална форма, в дизюнктивна нормална форма и като полиноми.

– Всеки алгоритъм, обработващ естествени числа, може да се реализира посредством компютърна програма, имаща единствено оператори за присвояване, оператори за четене и запис на входни и изходни данни, само един оператор за цикъл и никакви други оператори. Това е известно като нормална форма на Клѝни.

– В алгебрата всяка матрица може да се представи в жорданова нормална форма.

– Представянето на всяко естествено число като произведение на прости числа също може да се разглежда като нормална форма.

– В ламбда-смятането се дефинира понятието редукция на ламбда-термове. Когато един ламбда-терм бъде доведен посредством редукция до ламбда-терм, който не може повече да се редуцира, казваме, че сме получили бета нормална форма. Бета нормалната форма е резултатът от пресмятането (следователно зациклящите се изчислителни процеси не притежават бета нормална форма).

В този раздел ще дефинираме няколко нормални форми за формули класическата предикатна логика от първи ред.* Нещата, които са верни само при класическата логика, но не и при интуиционистката логика, ще бъдат отбелязвани с **(клас)**.

3.7.1. Твърдение. *За произволни формули φ и ψ:*

а) $\models \neg(\varphi \vee \psi) \Leftrightarrow \neg\varphi \,\&\, \neg\psi$

б) $\models \neg(\varphi \,\&\, \psi) \Leftrightarrow \neg\varphi \vee \neg\psi$** **(клас)**

в) $\models \neg(\varphi \Rightarrow \psi) \Leftrightarrow \varphi \,\&\, \neg\psi$*** **(клас)**

г) $\models \neg\neg\varphi \Leftrightarrow \varphi$ **(клас)**

Доказателство. **(а)** Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \models \varphi[v]$, а B е съждението $\mathbf{M} \models \psi[v]$.

Най-напред да докажем, че $\mathbf{M} \models \neg(\varphi \vee \psi) \Rightarrow \neg\varphi \,\&\, \neg\psi[v]$. За целта да допуснем, че не е вярно съждението „A или B“. Трябва да докажем, че не е вярно A и не е вярно B. За да докажем, че не е вярно A, да допуснем, че A е вярно. Щом A е вярно, значи е вярно и съждението

*Няма хубави нормални форми за интуиционистката предикатна логика.

**Интуиционистки е вярно $\models \neg(\varphi \,\&\, \psi) \Leftrightarrow \neg\neg(\neg\varphi \vee \neg\psi)$.

***Интуиционистки е вярно $\models \neg(\varphi \Rightarrow \psi) \Leftrightarrow \neg\neg\varphi \,\&\, \neg\psi$.

„A или B“. Но това е противоречие, значи A не е вярно. Аналогично се вижда, че и B не е вярно.

За да докажем, че $\mathbf{M} \vDash \neg\varphi \,\&\, \neg\psi \Rightarrow \neg(\varphi \vee \psi)[v]$, да допуснем, че не е вярно A и не е вярно B. Трябва да докажем, че не е вярна дизюнкцията „A или B“. За да докажем, че тази дизюнкция не е вярна, да допуснем, че тя е вярна и да разгледаме случаи в зависимост от това дали е вярно A или B. Когато е вярно A получаваме противоречие, защото знаем, че A не е вярно. Но когато е вярно B също получаваме противоречие, защото знаем, че B не е вярно. И в двата случая стигаме до противоречие. То се дължи на допускането, че дизюнкцията „A или B“ е вярна.

(б) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, а B е съждението $\mathbf{M} \vDash \psi[v]$.

Най-напред да докажем, че $\mathbf{M} \vDash \neg(\varphi \,\&\, \psi) \Rightarrow \neg\varphi \vee \neg\psi[v]$. За целта да допуснем, че не е вярно съждението „A и B“. Трябва да докажем, че A е невярно или B е невярно. От конструктивна гледна точка това означава, че ни е дадена функция, която получава аргументи от типове A и B и връща стойност от тип „противоречие“. Имайки само такава функция няма начин да направим функция, която получава само един аргумент от тип A и връща противоречие, нито пък функция, която получава само B и връща противоречие. Следователно няма как да конструираме обект от тип $\neg\varphi \vee \neg\psi$. Това ни подсказва, че ще трябва да допуснем противното.

И така, да допуснем, че не е вярно съждението „A не е вярно или B не е вярно“. Ако допуснем, че A е невярно, ще получим че е вярно съждението, за което току-що допуснахме, че е невярно. Значи съждението A не може да не е вярно. Аналогично се вижда, че и B не може да не е вярно. Това обаче противоречи на дизюнкцията, казваща, че A не е вярно или B не е вярно.

За да докажем обратната посока, да допуснем, че не е вярно A или не е вярно B. Трябва да докажем, че не е вярно съждението „A и B“. За целта да допуснем, че съждението „A и B“ е вярно. Значи съжденията A и B са верни, но това веднага ни дава противоречие с дизюнкцията, според която не е вярно A или не е вярно B.

(в) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, а B е съждението $\mathbf{M} \vDash \psi[v]$.

Най-напред да докажем, че $\mathbf{M} \vDash \neg(\varphi \Rightarrow \psi) \Rightarrow \varphi \,\&\, \neg\psi[v]$. За целта да допуснем, че не е вярно съждението „ако A, то B“. Трябва да докажем, че A е вярно и B е невярно. Ако допуснем, че A не е вярно, тогава

A ще ни даде противоречие, а значи и всичко. В частност получаваме, че от A следва B, а това дава противоречие, дължащо се на допускането, че A не е вярно. Интуиционистки това ни дава двойното отрицание на A, а класически — самото A. Сега да допуснем, че B е вярно. Тогава съждението „ако A, то B“ става тривиално вярно, а това ни дава противоречие. Както интуиционистки, така и класически това противоречие показва, че B не е вярно.

За да докажем обратната посока, да допуснем, че A е вярно и B е невярно. Трябва да докажем, че не е вярно съждението „ако A, то B“. Ами да допуснем, че то е вярно. Но ние знаем, че A е вярно, това ни дава B, т. е. противоречие. Както интуиционистки, така и класически противоречието показва, че съждението „ако A, то B“ не е вярно.

(г) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$. Трябва да докажем, че съждението „не не A“ еквивалентно на A. При използване на класическата логика това е тривиално вярно. ∎

3.7.2. Твърдение. *За произволна формула* φ

а) $\vDash \neg\exists \mathtt{x}\, \varphi \Leftrightarrow \forall \mathtt{x}\, \neg\varphi$

б) $\vDash \neg\forall \mathtt{x}\, \varphi \Leftrightarrow \exists \mathtt{x}\, \neg\varphi$[*] **(клас)**

Доказателство. **(а)** Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. За произволен елемент μ на универсума на A, нека $A(\mu)$ е съждението $\mathbf{M} \vDash \varphi[\mathtt{x} := \mu|v]$.

Най-напред ще докажем, че $\mathbf{M} \vDash \neg\exists \mathtt{x}\, \varphi \Rightarrow \forall \mathtt{x}\, \neg\varphi[v]$. За целта да допуснем, че не съществува $\mu \in |\mathbf{M}|$, за което $A(\mu)$ е истина. Трябва да докажем, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е лъжа. Да изберем произволно $\mu \in |\mathbf{M}|$ и да допуснем, че $A(\mu)$ е истина. Вижда се, че това противоречи на допускането, че не съществува такова μ.

За да докажем обратната посока, да допуснем, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е лъжа. Трябва да докажем, че не съществува $\mu \in |\mathbf{M}|$, за което $A(\mu)$ е истина. Да допуснем, че такова μ съществува. Тогава за това μ съждението $A(\mu)$ ще бъде истина, което противоречи на допускането, че за всяко μ съждението $A(\mu)$ е лъжа.

(б) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. За произволен елемент μ на универсума на A, нека $A(\mu)$ е съждението $\mathbf{M} \vDash \varphi[\mathtt{x} := \mu|v]$.

[*] Интуиционистки е вярно $\vDash \neg\forall \mathtt{x}\, \varphi \Leftrightarrow \neg\neg\exists \mathtt{x}\, \neg\varphi$.

Най-напред ще докажем, че $\mathbf{M} \vDash \neg\forall \mathtt{x}\, \varphi \Rightarrow \exists \mathtt{x}\, \neg\varphi[v]$. Да допуснем, че не е вярно, че за всеки елемент μ на универсума на $\mathbf{M}$ е вярно $A(\mu)$. От конструктивна гледна точка това означава, че ни е дадена функция, която ще ни върне обект от тип противоречие, ако ѝ дадем като аргумент функция, която за всяко μ връща обект от тип $A(\mu)$. Не се вижда как, разполагайки с такава функция, ще можем да намерим обект μ, за който не е вярно $A(\mu)$. Следователно съждението не може да се докаже конструктивно и значи трябва да допуснем противното.

И така, да допуснем, че не съществува $\mu \in |\mathbf{M}|$, за което не е вярно $A(\mu)$. Използвайки доказаното в а), можем да заключим, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е вярно.* Това е противоречие.

За да докажем обратната посока, да допуснем, че не съществува елемент μ на универсума на $\mathbf{M}$, за който е вярно съждението $A(\mu)$. Трябва да докажем, че за всеки елемент μ на универсума на $\mathbf{M}$, съждението $A(\mu)$ е невярно. Това е очевидно. ∎

3.7.3. Твърдение. *За произволни формули φ и ψ:*

$$\vDash (\varphi \Rightarrow \psi) \Leftrightarrow (\neg\varphi \vee \psi)^{**} \qquad \textbf{(клас)}$$

Доказателство. Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, а B е съждението $\mathbf{M} \vDash \psi[v]$.

Най напред ще докажем, че $\mathbf{M} \vDash (\varphi \Rightarrow \psi) \Rightarrow (\neg\varphi \vee \psi)[v]$. Да допуснем, че от A следва B. Трябва да докажем, че A не е вярно или B е вярно. От конструктивна гледна точка това означава, че имаме функция, на която ако ѝ дадем аргумент от тип A, ще ни върне стойност от тип B. Не се вижда как имайки такава функция това ще ни позволи да установим кой от членовете на исканата дизюнкция е верен. Това ни подсказва, че желаното свойство не може да бъде доказано конструктивно и значи трябва да допуснем противното.

И така, да допуснем, че не е вярна дизюнкцията, казваща, че A е невярно или B е вярно. Това означава, че A е истина и B е лъжа (вж. доказателството на твърдение 3.7.1 а)). Но щом A е истина и от A следва B, значи B е истина. Стигнахме до противоречие.

За да докажем обратната посока, да допуснем, че A е невярно или B е вярно. Трябва да докажем, че от A следва B. За целта да допуснем, че A е вярно. Щом A е вярно, то значи дизюнкцията „A е невярно или

*Всъщност от доказаното в а) можем да заключим, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ не може да не е вярно. Използвайки класическата логика, от тук, разбира се, получаваме, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е вярно.

**Интуиционистки не е възможно да изразим импликацията посредством останалите съждителни операции. Вярно е обаче следното: $\vDash \neg\neg(\varphi \Rightarrow \psi) \Leftrightarrow \neg\neg(\neg\varphi \vee \psi)$.

B е вярно“ ни казва, че B е вярно. И така, допуснахме, че A е вярно и доказахме, че B е вярно. Следователно от A следва B. ∎

3.7.4. Преди много хилядолетия в град Лагаш, намиращ се в днешен Ирак, живял бог Нингирсу, който бил юначен ловец и воин. Веднъж, по време на един от ловните си походи, Нингирсу влязал в бой с многоглавия дракон Мушмаху. Всеки път когато Нингирсу отрязвал една от главите на Мушмаху, на него му пораствали много нови глави. Отначало Нингирсу решил, че работата е безнадеждна, но после забелязал, че всеки път когато отрязвал някоя глава, шиите на новопоникналите глави били най-малко с един сантиметър по-къси. Понеже разбирал от математика, Нингирсу съобразил, че ако не се отказва, със сигурност ще се стигне до момент, когато драконът ще остане без глави. След като победил, Нингирсу се отправил към град Нипур, където се намирал зигуратът Екур — жилището на боговете. Нингирсу бил изпълнен с боен плам, надавал викове, пеел песни, с които прославял геройството си, и извършвал разни вандалства с цел да му обръщат внимание. Боговете в Нипур се уплашили и му казали, че ако се поуспокои малко, ще го възнаградят. Като стигнал Екур, той показал на удивените богове бойните си трофеи.

Логиците използват т. н. *ординални числа* с цел да „измерят“ колко е сложно дадено разсъждение, използващо индукция. Ако използваме обикновената индукция за естествени числа по възможно най-естествения начин, казваме, че сме използвали индукция до ординала ω. Ако обаче докато правим индукционната стъпка, вътре в нея за втори път започваме да правим индукция, казваме че сме използвали индукция до ординала ω^2. Ако пък и в индукционната стъпка на втората индукция пак правим индукция, тогава използваме индукция до ω^3. Според „великото предположение“ на Харви Фридман, всяка теорема, публикувана в математическо списание като „Annals of Mathematics“, чиято формулировка използва само крайни обекти, може да се докаже с индукция до ω^3. Въпреки това, оказва се, че само с индукция до ω^3 не можем да докажем, че драконът Мушмаху ще бъде победен. За целта ни е нужна индукция до $\omega^\omega = \sup\{\omega, \omega^2, \omega^3, \omega^4, \dots\}$.

Въпреки че драконът Мушмаху бил убит, той си оставил достоен потомък — Лернейската хидра. Също както и при Мушмаху, и на Лернейската хидра за всяка отсечена глава ѝ порастват нови глави. Новите глави на Хидрата обаче растяли по малко по-различни правила, така че понякога се случвало шиите на новите глави да не са по-къси от шията на току-що отсечената глава. Хидрата била убита от Херкулес, защото отново се оказало, че без значение как ѝ режем главите,

след краен брой стъпки тя ще остане без глави. Това обаче е изключително трудно да се докаже, защото изисква индукция до ординала $\varepsilon_0 = \sup\{\omega, \omega^\omega, \omega^{\omega^\omega}, \omega^{\omega^{\omega^\omega}}, \dots\}$.* Оказва се, че в стандартната аксиоматична теория на аритметиката — Аритметиката на Пеано — е невъзможно да правим толкова сложни индукции и затова в тази теория не може да се докаже, че Хидрата ще бъде убита от Херкулес.

В следващата дефиниция да си мислим, че редицата от числа $(a_n)_{n=1}^\infty$ измерва броя на главите на дракона Мушмаху. Числото a_1 е равно на броя на главите, чиято шия е дълга 1 сантиметър, a_2 е равно на броя на главите, чиято шия е равна на 2 сантиметра и т. н.

3.7.5. Дефиниция. Нека $(a_n)_{n=1}^\infty$ е редица от естествени числа и редицата $(b_n)_{n=1}^\infty$ се получава като извършим следните промени в редицата $(a_n)_{n=1}^\infty$:

- избираме такова естествено число i, че $a_i \neq 0$;
- намаляваме a_i;
- заменяме $a_1, a_2, \dots, a_{i-1}$ с произволни естествени числа.

В такъв случай казваме, че редицата $(a_n)_{n=1}^\infty$ се *конвертира* в редицата $(b_n)_{n=1}^\infty$. Също ще казваме, че редицата $(b_n)_{n=1}^\infty$ се получава от редицата $(a_n)_{n=1}^\infty$ посредством *конверсия*. Числото i ще наричаме *ранг на конверсията*.

3.7.6. Лема за фундираност на конверсията. *Нека $(a_n)_{n=1}^\infty$ е редица от естествени числа, в която има само краен брой ненулеви елементи. Да разгледаме следния процес: прилагаме конверсия към редицата $(a_n)_{n=1}^\infty$, после прилагаме конверсия към новополучената редица, след това отново прилагаме конверсия и т. н. Без значение какви точно конверсии извършваме, със сигурност след краен брой стъпки ще стигнем до редица, съдържаща само нули.*

Доказателство. Да забележим, че не е възможно да прилагаме безброй много конверсии, чийто ранг е 1. Наистина, след всяка такава конверсия първият член на редицата намалява, а това не може да се случва до безкрайност. Това означава, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне 2.

*Всеки може сам да се постави на мястото на Херкулес и да се опита да победи Лернейската хидра, използвайки джава аплет от адрес `http://math.andrej.com/2008/02/02/the-hydra-game/`.

Да забележим също, че не е възможно да прилагаме безброй много конверсии, чийто ранг е 1 или 2. Наистина, вече видяхме, че не можем да прилагаме безброй конверсии само от ранг 1, следователно ако прилагаме само конверсии от ранг 1 или 2, ще трябва да прилагаме безброй пъти конверсии от ранг 2. Това обаче е невъзможно, защото след всяка конверсия от ранг 2 вторият член на редицата намалява, а това не може да се случва до безкрайност (да забележим, че конверсиите от ранг 1 не променят втория член на редицата). Това означава, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне 3.

Обаче не е възможно също да прилагаме безброй много конверсии, чийто ранг е 1, 2 или 3. Наистина, вече видяхме, че не можем да прилагаме безброй конверсии само от рангове 1 или 2, следователно ако прилагаме само конверсии от ранг 1, 2 или 3, ще трябва да прилагаме безброй пъти конверсии от ранг 3. Това обаче е невъзможно, защото след всяка конверсия от ранг 3 третият член на редицата намалява, а това не може да се случва до безкрайност (да забележим, че конверсиите от рангове 1 или 2 не променят третия член на редицата). Това означава, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне 4.

Разсъждавайки по подобен начин, можем да стигнем до извода, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне k, където k е произволно предварително избрано естествено число. Нека числото k е такова, че всички ненулеви членове в $(a_n)_{n=1}^{\infty}$ са измежду $a_1, a_2, \ldots, a_{k-1}$. В такъв случай няма да бъде възможна конверсия от ранг k или по-голям, следователно както и да прилагаме конверсиите, със сигурност ще стигнем до редица, съдържаща само нули. ■

✓ **3.7.7. Дефиниция.** Казваме, че дадена формула е в *отрицателна нормална форма*, ако:

- във формулата няма други логически операции, освен конюнкция ($\&$), дизюнкция ($\vee$), отрицание ($\neg$) и кванторите за всеобщност ($\forall$) и съществуване ($\exists$);
- всички отрицания във формулата се намират пред атомарни подформули.

✓ **3.7.8. Твърдение.** *За всяка формула може да се намери еквивалентна на нея (според класическата логика) формула, която е в отрицателна нормална форма.*

✓ Доказателство. За краткост, навсякъде в това доказателство където се говори за импликации, ще имаме предвид импликации, които не са отрицания, т. е. от вида $\varphi \Rightarrow \psi$, където $\psi \neq \bot$.

Нека φ е произволна формула. Най-напред да елиминираме импликациите във φ по следния начин: да заменяме докато може всяка подформула във φ, имаща вида $\psi' \Rightarrow \psi''$, където $\psi'' \neq \bot$, с формулата $\neg\psi' \vee \psi''$. Съгласно твърдение 3.7.3 след всяка такава замяна получаваме еквивалентна формула. Тъй като след всяка замяна броят на импликациите, намалява, то след краен брой стъпки ще получим формула без импликации, което значи, че във формулата няма други логически операции, освен конюнкция ($\&$), дизюнкция ($\vee$), отрицание ($\neg$) и кванторите за всеобщност ($\forall$) и съществуване ($\exists$).

Остава да „преместим“ отрицанията пред атомарни формули. За целта да прилагаме докато може замени от следния вид:

$$\neg\neg\psi \longmapsto \psi \tag{12}$$
$$\neg(\psi' \vee \psi'') \longmapsto \neg\psi' \,\&\, \neg\psi'' \tag{13}$$
$$\neg(\psi' \,\&\, \psi'') \longmapsto \neg\psi' \vee \neg\psi'' \tag{14}$$
$$\neg\exists \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, \neg\psi \tag{15}$$
$$\neg\forall \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, \neg\psi \tag{16}$$

Съгласно твърдения 3.7.1 г), 3.7.1 а), 3.7.1 б), 3.7.2 а) и 3.7.2 б), след всяка такава замяна получаваме еквивалентна формула. Ако след краен брой стъпки получим формула, към която не можем да приложим никоя от тези замени, то това означава, че сме стигнали до формула в отрицателна нормална форма. Това ни дава т. н. частична коректност на алгоритъма за привеждане в отрицателна нормална форма. Остава да видим защо този алгоритъм никога не се зацикля, т.е. трябва да докажем, че можем да прилагаме замени от горния вид само краен брой пъти. Това ще направим по два начина.

(I начин) Ще използваме фундираността на конверсията (лема 3.7.6).

Нека χ е произволна формула. *Височина* на χ ще наричаме дължината на най-дългата редица от вида

$$\chi_1, \chi_2, \chi_3, \ldots, \chi_k$$

където $\chi_1 = \chi$, $\chi_i \neq \chi_{i+1}$ и χ_{i+1} е подформула на χ_i. Ако си мислим формулата като дърво, тогава височината ѝ е равна на дължината на най-дългия клон в това дърво.

За всяка формула φ ще дефинираме редица $(a_n)_{n=1}^{\infty}$, която ще наречем *редица на* φ, по следния начин: нека a_i е равно на броя на подформулите на φ, които са от вида $\neg\chi$ и са с височина точно i.

Може да се забележи, че ако формулата φ' се получава от φ посредством някоя от замените (12)–(16), тогава редицата на φ' може да се получи от редицата на φ посредством конверсия. Съгласно лемата за фундираност на конверсията, след краен брой стъпки или ще стигнем формула, към която не можем да приложим никоя от замените (12)–(16), или ще стигнем формула, чиято редица се състои само от нули. Последното би означавало, че във формулата няма нито едно отрицание, но в този случай също е невъзможно да приложим замените (12)–(16).

(II начин) Да разгледаме следния начин, по който от формула получаваме аритметичен израз. Да заменим всяка атомарна формула с числото две. Да заменим всяка подформула от вида $\psi' \,\&\, \psi''$, $\psi' \vee \psi''$ и $\psi' \Rightarrow \psi''$ с $x + y$, където x и y са изразите, получени съответно от ψ' и ψ''. Да заменим всяка подформула от вида $\neg\psi$ с x^2, където x е изразът, получен от ψ. Да заменим всяка подформула от вида $\forall \mathtt{x}\, \psi$ и $\exists \mathtt{x}\, \psi$ с $1 + x$, където x е изразът, получен от ψ. Може да се забележи, че така полученият израз е естествено число, не по-малко от 2. Тъй като за всеки естествени числа по-големи или равни на 2 е изпълнено $(x^2)^2 > x$, $(x + y)^2 > x^2 + y^2$ и $(1 + x)^2 > 1 + x^2$, то стойността на получения израз ще намалява след всяка замяна от горния вид, а това не може да продължи до безкрайност. ∎

3.7.9. ***Забележка:*** Когато използваме първия начин за да докажем, че алгоритъмът за привеждане в отрицателна нормална форма спира, не е нужно да се използва толкова сложно твърдение като фундираността на конверсията. Нека редицата, съответстваща на някоя формула е $(a_n)_{n=1}^{\infty}$ и да разгледаме числото

$$\sum_{i=1}^{\infty} a_i 3^i$$

(сумата е коректна, защото само краен брой от числата a_i са ненулеви). Докато при дракона Мушмаху когато някоя негова глава бъде отсечена, може да му пораснат произволен брой нови глави, то тук, когато

вкараме някое отрицание „навътре“ във формулата, то се заменя най-много с две „по-малки“ отрицания. Това означава, че след всяка замяна от вида (12)–(16), така дефинираното число ще намалява, а това не може да продължи до безкрайност.

3.7.10. Твърдение. *За произволни формули φ и ψ, ако* $\mathtt{x}$ *не е свободна променлива на φ, то:*

а) $\models \varphi \,\&\, \exists \mathtt{x}\, \psi \Leftrightarrow \exists \mathtt{x}\, (\varphi \,\&\, \psi)$ $u \models \exists \mathtt{x}\, \psi \,\&\, \varphi \Leftrightarrow \exists \mathtt{x}\, (\psi \,\&\, \varphi)$;

б) $\models \varphi \,\&\, \forall \mathtt{x}\, \psi \Leftrightarrow \forall \mathtt{x}\, (\varphi \,\&\, \psi)$ $u \models \forall \mathtt{x}\, \psi \,\&\, \varphi \Leftrightarrow \forall \mathtt{x}\, (\psi \,\&\, \varphi)$;

в) $\models \varphi \vee \exists \mathtt{x}\, \psi \Leftrightarrow \exists \mathtt{x}\, (\varphi \vee \psi)$ $u \models \exists \mathtt{x}\, \psi \vee \varphi \Leftrightarrow \exists \mathtt{x}\, (\psi \vee \varphi)$;

г) $\models \varphi \vee \forall \mathtt{x}\, \psi \Leftrightarrow \forall \mathtt{x}\, (\varphi \vee \psi)$ $u \models \forall \mathtt{x}\, \psi \vee \varphi \Leftrightarrow \forall \mathtt{x}\, (\psi \vee \varphi)$. **(клас)**

Доказателство. **(а)** Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \models \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \models \psi[\mathtt{x} := \mu|v]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемнт μ на универсума на $\mathbf{M}$, съждението $\mathbf{M} \models \varphi[\mathtt{x} := \mu|v]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \models \varphi \,\&\, \exists \mathtt{x}\, \psi \Rightarrow \exists \mathtt{x}\, (\varphi \,\&\, \psi)[v]$. Да допуснем, че съждението A е вярно и съществува такова $\mu \in |\mathbf{M}|$, че съждението $B(\mu)$ е вярно. В такъв случай очевидно съществува такова $\mu \in |\mathbf{M}|$, че съждението „A и $B(\mu)$“ е вярно.

За да докажем обратната посока, да допуснем, че съществува такова $\mu \in |\mathbf{M}|$, че съждението „A и $B(\mu)$“ е вярно. В такъв случай очевидно съждението A ще бъде вярно и освен това ще съществува $\mu \in |\mathbf{M}|$, за което съждението $B(\mu)$ е вярно.

(б) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \models \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \models \psi[\mathtt{x} := \mu|v]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемнт μ на универсума на $\mathbf{M}$, съждението $\mathbf{M} \models \varphi[\mathtt{x} := \mu|v]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \models \varphi \,\&\, \forall \mathtt{x}\, \psi \Rightarrow \forall \mathtt{x}\, (\varphi \,\&\, \psi)[v]$. Да допуснем, че е вярно A и че за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. В такъв случай очевидно за всяко $\mu \in |\mathbf{M}|$ ще бъде вярно съждението „A и $B(\mu)$“.

За да докажем обратната посока, да допуснем, че за всяко $\mu \in |\mathbf{M}|$ е вярно съждението „A и $B(\mu)$“. Тъй като универсумът на $\mathbf{M}$ е непразен, като приложим току-що допуснатото за някой елемент на универсума, ще получим, че съждението A е вярно. Освен това очевидно за всяко $\mu \in |\mathbf{M}|$ ще бъде вярно съждението $B(\mu)$.

(в) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \vDash \psi[\mathtt{x} := \mu|v]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемнт μ на универсума на $\mathbf{M}$, съждението $\mathbf{M} \vDash \varphi[\mathtt{x} := \mu|v]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \vDash \varphi \vee \exists \mathtt{x}\, \psi \Rightarrow \exists \mathtt{x}\, (\varphi \vee \psi)[v]$. Да допуснем, че е вярно A или за някое $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. В първия случай, когато е вярно A, ще съществува $\mu \in |\mathbf{M}|$, за което е вярно съждението „A или $B(\mu)$“, защото кой да е елемент μ на универсума ще ни свърши работа (а такъв има, защото универсъмът е непразно множество). Във втория случай, когато съществува $\mu \in |\mathbf{M}|$, за което е вярно $B(\mu)$, очевидно също ще съществува $\mu \in |\mathbf{M}|$, за което е вярно съждението „A или $B(\mu)$“.

За да докажем обратната посока, да допуснем, че съществува $\mu \in |\mathbf{M}|$, за което е вярно съждението „A или $B(\mu)$“. Ако за това μ, е вярно A, то A е вярно (това съждение не зависи от μ). Ако пък за съществуващото μ е вярно $B(\mu)$, то значи съществува $\mu \in |\mathbf{M}|$, за което е вярно $B(\mu)$. Следователно A е вярно или съществува $\mu \in |\mathbf{M}|$, за което е вярно $B(\mu)$.

(г) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \vDash \psi[\mathtt{x} := \mu|v]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемент μ на универсума на $\mathbf{M}$, съждението $\mathbf{M} \vDash \varphi[\mathtt{x} := \mu|v]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \vDash \varphi \vee \forall \mathtt{x}\, \psi \Rightarrow \forall \mathtt{x}\, (\varphi \vee \psi)[v]$. За целта да допуснем, че е вярно A или за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. В първия случай очевидно за всяко $\mu \in |\mathbf{M}|$ ще бъде вярно съждението „A или $B(\mu)$“. Във втория случай — също.

За да докажем обратната посока, да допуснем, че за всяко $\mu \in |\mathbf{M}|$ е вярно съждението „A или $B(\mu)$“. От конструктивна гледна точка това означава, че ни е дадена функция, която по дадено μ ни казва дали е вярно A или $B(\mu)$. Ако решим „да пробваме“ тази функция за различни елементи на универсума и всеки път получаваме, че е вярно $B(\mu)$, това няма да означава, че $B(\mu)$ е вярно винаги — възможно е просто да не сме имали късмет и ако бяхме улучили „правилното“ μ, щяхме да установим, че е вярно A. Следователно няма как имайки такава функция

да разберем дали е вярно A, или за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. Това ни подсказва, че съждението не е вярно конструктивно и значи трябва да допуснем противното.

И така, да допуснем, че не е вярно съждението „A е вярно или за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$“. Това означава (вж. доказателството на съждение 3.7.1 а)), че съждението A не е вярно и съждението „за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$“ също не е вярно. Но ние знаем, че за всяко $\mu \in |\mathbf{M}|$ е вярно „A или $B(\mu)$“. Тъй като току-що установихме, че A не е вярно, то значи за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. Това е противоречие. ■

3.7.11. Твърдение. *За произволни формули φ и ψ, ако $\mathtt{x}$ не е свободна променлива на φ, то:*

а) $\models (\varphi \Rightarrow \forall \mathtt{x}\, \psi) \Leftrightarrow \forall \mathtt{x}\, (\varphi \Rightarrow \psi)$;

б) $\models (\varphi \Rightarrow \exists \mathtt{x}\, \psi) \Leftrightarrow \exists \mathtt{x}\, (\varphi \Rightarrow \psi)$; **(клас)**

в) $\models (\exists \mathtt{x}\, \psi \Rightarrow \varphi) \Leftrightarrow \forall \mathtt{x}\, (\psi \Rightarrow \varphi)$;

г) $\models (\forall \mathtt{x}\, \psi \Rightarrow \varphi) \Leftrightarrow \exists \mathtt{x}\, (\psi \Rightarrow \varphi)$. **(клас)**

✓ **3.7.12. Дефиниция.** Казваме, че една формула е в *пренексна нормална форма*, ако тя има вида

$$\mathtt{Я}_1 \mathtt{x}_1 \mathtt{Я}_2 \mathtt{x}_2 \ldots \mathtt{Я}_n \mathtt{x}_n (\varphi)$$

където $\mathtt{Я}_1, \mathtt{Я}_2, \ldots, \mathtt{Я}_n$ са квантори ($\forall$ или $\exists$), а формулата φ не съдържа квантори.

✓ **3.7.13. Твърдение.** *За всяка формула може да се намери еквивалентна на нея (според класическата логика) формула, която е в пренексна нормална форма.*

✓ Доказателство. Нека ни е дадена произволна формула. Съгласно лема 2.6.11 може да намерим конгруентна, а значи и еквивалентна на нея формула, в която всички квантори имат различни променливи и никоя кванторна променлива не е свободна променлива във формулата. Да прилагаме в така получената формула докато може замени от

следния вид:

$$\begin{gathered}
\varphi \,\&\, \exists \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, (\varphi \,\&\, \psi) \\
\exists \mathtt{x}\, \psi \,\&\, \varphi \longmapsto \exists \mathtt{x}\, (\psi \,\&\, \varphi) \\
\varphi \,\&\, \forall \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, (\varphi \,\&\, \psi) \\
\forall \mathtt{x}\, \psi \,\&\, \varphi \longmapsto \forall \mathtt{x}\, (\psi \,\&\, \varphi) \\
\varphi \vee \exists \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, (\varphi \vee \psi) \\
\exists \mathtt{x}\, \psi \vee \varphi \longmapsto \exists \mathtt{x}\, (\psi \vee \varphi) \\
\varphi \vee \forall \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, (\varphi \vee \psi) \\
\forall \mathtt{x}\, \psi \vee \varphi \longmapsto \forall \mathtt{x}\, (\psi \vee \varphi) \\
\varphi \Rightarrow \forall \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, (\varphi \Rightarrow \psi) \\
\varphi \Rightarrow \exists \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, (\varphi \Rightarrow \psi)
\end{gathered}$$

$$\begin{gathered}
\exists \mathtt{x}\, \psi \Rightarrow \varphi \longmapsto \forall \mathtt{x}\, (\psi \Rightarrow \varphi) \\
\forall \mathtt{x}\, \psi \Rightarrow \varphi \longmapsto \exists \mathtt{x}\, (\psi \Rightarrow \varphi) \\
\neg \exists \mathtt{x}\, \varphi \longmapsto \forall \mathtt{x}\, \neg \varphi \\
\neg \forall \mathtt{x}\, \varphi \longmapsto \exists \mathtt{x}\, \neg \varphi
\end{gathered}$$

Ако имаме подформула от вида напр. $\varphi \,\&\, \exists \mathtt{x}\, \psi$, то във φ променливата $\mathtt{x}$ няма да бъде свободна, защото ако $\mathtt{x}$ бе свободна във φ, то тъй като работим с формула, в която няма два квантора с една и съща променлива, то променливата $\mathtt{x}$ би била свободна и в цялата формула, а пък си осигурихме да няма кванторна променлива, която да е свободна в цялата формула. Това означава, че съгласно твърдения 3.7.10 б), 3.7.10 а), 3.7.10 г), 3.7.10 в), 3.7.11 а), 3.7.11 б), 3.7.11 в), 3.7.11 г), 3.7.2 а) и 3.7.2 б) при всяка една от тези замени ще получаваме еквивалентни формули. Ако след краен брой стъпки стигнем формула, към която не може да прилагаме никоя от по-горните замени, то значи сме получили формула в пренексна нормална форма. Това ни дава частичната коректност на алгоритъма за привеждане в пренексна нормална форма. Остава да докажем, че алгоритъмът винаги спира.

(I начин) За произволна формула χ да наречем *дълбочина* на подформулата χ' на χ дължината на най-дългата редица от вида

$$\chi_1, \chi_2, \chi_3, \ldots, \chi_k$$

където $\chi_1 = \chi$, $\chi_k = \chi'$, $\chi_i \neq \chi_{i+1}$, χ_{i+1} е подформула на χ_i и формулите $\chi_1, \chi_2, \ldots, \chi_{k-1}$ не започват с квантор. Ако си мислим формулата като

дърво, тогава дълбочината на една подформула е равна на разстоянието между корена на поддървото и корена на цялото дърво, като при това пропускаме възлите от дървото, в които стои квантор.

Да забележим, че замените от горния вид не променят броя на подформулите, започващи с квантор. Всяка една такава замяна обаче намалява дълбочината на някоя от подформулите, започващи с квантор, а това не може да продължи до безкрайност.

(II начин) Да разгледаме следния начин, по който от формула получаваме аритметичен израз. Да заменим всяка атомарна подформула с числото 2. Да заменим всяка подформула от вида $\chi' \,\&\, \chi'$, $\chi' \vee \chi''$ и $\chi' \Rightarrow \chi''$ с $x + y$, където x и y са изразите, получени съответно от χ' и χ''. Да заменим всяка подформула от вида $\neg\chi$ с $1 + x$, където x е изразът, получен от χ. Да заменим всяка подформула от вида $\forall \mathtt{x}\, \chi$ и $\exists \mathtt{x}\, \chi$ с x^2, където x е изразът, получен от χ. Може да се забележи, че така полученият израз е естествено число, не по-малко от 2. Тъй като за всеки естествени числа, не по-малки от 2, е изпълнено $x + y^2 < (x + y)^2$, $x^2 + y < (x + y)^2$ и $1 + x^2 < (1 + x)^2$, то след всяко преобразование на формулата по описание по-горе начин, стойността на съответния аритметичен израз ще се увеличи. Това обаче не може да продължи до безкрай, защото броят на символите в тези аритметични изрази се запазва след всяко такова преобразование и значи може да се получат само краен брой аритметични изрази. ∎

3.7.14. ***Забележка*:** Въпреки че правилата за замяна в алгоритъма за привеждане в пренексна нормална форма са много на брой, те се помнят лесно. Нека да забележим, че когато кванторът излиза пред отрицание или се е намирал отляво на импликация, той се променя, т.е. от $\forall$ става $\exists$ и от $\exists$ става $\forall$. Във всички останали случаи кванторът се премества без промяна.

✓ **3.7.15. Твърдение.** *За всяка формула може да се намери еквивалентна на нея (според класическата логика) формула, която е едновременно в пренексна нормална форма и в отрицателна нормална форма.*

✓ Доказателство. Непосредствено се проверява, че ако преобразуваме според алгоритъма за привеждане в пренексна нормална форма формула, която се намира в отрицателна нормална форма, то ще получаваме формули в отрицателна нормална форма. Това означава, че можем просто да приведем формулата в отрицателна нормална фор-

ма и след това да приложим алгоритъмът за привеждане в пренексна нормална форма.

Също така можем непосредствено да проверим, че ако преобразуваме според алгоритъма за привеждане в отрицателна нормална форма формула, която се намира в пренексна нормална форма, то ще получаваме формули в пренексна нормална форма. Това означава, че можем просто да приведем формулата в пренексна нормална форма и след това да приложим алгоритъмът за привеждане в отрицателна нормална форма.

Също така, напълно допустимо е да прилагаме разбъркано замените от двата алгоритъма. В този случай процесът също със сигурност ще бъде краен и ще даде желания резултат, но тук няма да доказваме това. ∎

3.7.16. ***Забележка*:** Ако формулата вече е приведена в отрицателна нормална форма и приложим към нея алгоритъма за привеждане в пренексна нормална форма, няма да ни се наложи да прилагаме нито веднъж замени, при които кванторът се променя, т.е. от $\forall$ става $\exists$ или от $\exists$ става $\forall$. Това означава, че не е нужно да работим стъпка по стъпка, а можем просто да преместим наведнъж всички квантори отпред на формулата (обаче без да променяме реда им!) и ще получим пренексна нормална форма.

✓ **3.7.17. Твърдение.** *Формула от вида* $\forall \mathtt{x}\,(\varphi)$ *е тъждествено вярна в някоя структура* $\mathbf{M}$ *тогава и само тогава, когато в* $\mathbf{M}$ *е тъждествено вярна формулата* φ.

✓ Доказателство. Да допуснем, че $\mathbf{M} \vDash \forall \mathtt{x}\,(\varphi)$, т.е. за всяка оценка v в $\mathbf{M}$ е вярно $\mathbf{M} \vDash \forall \mathtt{x}\,(\varphi)[v]$. По дефиниция това означава, че за всяка оценка v в $\mathbf{M}$ и за всеки елемент μ на универсума на $\mathbf{M}$ е вярно $\mathbf{M} \vDash \varphi[\mathtt{x} := \mu|v]$. В частност, ако изберем $\mu = v(\mathtt{x})$, оценката $[\mathtt{x} := \mu|v]$ ще съвпадне с v и значи за всяка оценка v в $\mathbf{M}$ ще бъде вярно $\mathbf{M} \vDash \varphi[v]$. Следователно φ е тъждествено вярна в $\mathbf{M}$.

Обратната посока се вижда още по-лесно. Да допуснем, че φ е тъждествено вярна в $\mathbf{M}$. Това значи, че φ ще бъде вярна при произволна оценка. В частност φ ще бъде вярна и при всяка модифицирана оценка $[\mathtt{x} := \mu|v]$, и значи формулата $\forall \mathtt{x}\,\varphi$ е тъждествено вярна в $\mathbf{M}$. ∎

Прилагателното „скулемов“ от следващата дефиниция произлиза от името на откривателя на преобразованието, наречено *скулемизация* — норвежкия логик Тӯралф Скӯлем.

При скулемизацията ще използваме субституции, които променят една единствена променлива. За удобство ще използваме следното означение (подобно на означението, което имаме за оценки): ако $\mathtt{x}$ е променлива, а τ — терм, то с $\mathtt{x} \coloneqq \tau$ ще означаваме субституцията, която заменя $\mathtt{x}$ с τ и оставя всички останали променливи непроменени.

✓ **3.7.18. Дефиниция.** а) Нека е дадена формула от вида $\exists \mathtt{x}\,(\varphi)$, в която няма свободни променливи, а $\mathtt{c}$ е символ за константа. Формулата $\varphi[\mathtt{x} \coloneqq \mathtt{c}]$ се нарича *скулемово усилване* (от първи вид) на $\exists \mathtt{x}\,(\varphi)$.

б) Нека е дадена формула от вида $\exists \mathtt{x}\,(\varphi)$, чиито свободни променливи са $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$, а $\mathtt{f}$ е n-местен функционален символ. Формулата $\varphi[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)]$ се нарича *скулемово усилване* (от втори вид) на $\exists \mathtt{x}\,(\varphi)$.

в) Символът за константа $\mathtt{c}$ от а) и функционалният символ $\mathtt{f}$ от б) се наричат *скулемов символ за константа* и *скулемов функционален символ*.

3.7.19. Пример. Да разгледаме формулата $\varphi = \exists \mathtt{x}\,\mathtt{p}(\mathtt{x})$. Едно нейно скулемово усилване е формулата $\varphi' = \mathtt{p}(\mathtt{c})$. Във всяка структура $\mathbf{M}$ формулата φ „казва“, че в универсума на $\mathbf{M}$ има елемент, за който предикатът $\mathtt{p}^{\mathbf{M}}$ е истина. Формулата φ' пък казва, че предикатът $\mathtt{p}^{\mathbf{M}}$ е истина не за някой неопределен елемент, а за $\mathtt{c}^{\mathbf{M}}$. Следователно скулемовото усилване φ' казва повече, отколкото φ и ако формулата φ' е вярна в някоя структура, то и φ ще бъде вярна. По-долу твърдение 3.7.21 ще покаже, че това се случва винаги, а не само при този конкретен пример.

Ако имаме право сами да решим каква да бъде интерпретацията на символа $\mathtt{c}$ и формулата φ е вярна в $\mathbf{M}$, то ще можем да интерпретираме $\mathtt{c}$ така, че $\mathtt{c}^{\mathbf{M}}$ да бъде оня елемент от универсума, чието съществуване се твърди от формулата φ. Ако променим $\mathbf{M}$ по този начин, формулата φ' ще стана вярна. По-долу твърдение 3.7.22 ще покаже, че това се случва винаги при скулемово усилване от първи вид, а не само при този конкретен пример.

3.7.20. Пример. Да разгледаме формулата $\varphi = \exists \mathtt{y}\,\mathtt{p}(\mathtt{x}, \mathtt{y})$. Едно нейно скулемово усилване е формулата $\varphi' = \mathtt{p}(\mathtt{x}, \mathtt{f}(\mathtt{x}))$. Във всяка структура $\mathbf{M}$ формулата φ „казва“, че за всеки елемент μ на универсума на $\mathbf{M}$ съществува такъв елемент ν, че $\mathtt{p}^{\mathbf{M}}(\mu, \nu)$ е истина. Формулата φ' пък казва, че $\mathtt{p}^{\mathbf{M}}(\mu, \nu)$ е истина не за някое неопределено ν, а за $\nu = \mathtt{f}^{\mathbf{M}}(\mu)$. Следователно скулемовото усилване φ' казва повече, отколкото φ и ако

формулата φ' е вярна в някоя структура, то и φ ще бъде вярна. По-долу твърдение 3.7.21 ще покаже, че това се случва винаги, а не само при този конкретен пример.

Ако имаме право сами да решим каква да бъде интерпретацията на символа $\mathtt{f}$ и формулата φ е вярна в $\mathbf{M}$, то ще можем да интерпретираме $\mathtt{f}$ така, че $\mathtt{f}^{\mathbf{M}}(\mu)$ да бъде оня елемент ν от универсума, чието съществуване се твърди от формулата φ. Ако променим $\mathbf{M}$ по този начин, формулата φ' ще стана вярна. По-долу твърдение 3.7.22 ще покаже, че това се случва винаги при скулемово усилване от втори вид, а не само при този конкретен пример.

Да припомним твърдение 3.3.17, съгласно което за произволни формула φ със свободни променливи $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$, термове $\tau_1, \tau_2, \ldots, \tau_n$ и оценка v в структура $\mathbf{M}$

$$\mathbf{M} \vDash \varphi[\tau_1, \ldots, \tau_n][v] \longleftrightarrow \mathbf{M} \vDash \varphi[[\![\tau_1]\!]^{\mathbf{M}} v, \ldots, [\![\tau_n]\!]^{\mathbf{M}} v] \tag{17}$$

Следващото твърдение обяснява защо скулемовото усилване е наречено „усилване“ — скулемовото усилване на една формула винаги казва повече неща, отколкото самата формула.

✓ **3.7.21. Твърдение.** *Ако скулемовото усилване на една формула е тъждествено вярно в структура* $\mathbf{M}$*, то и самата формула е тъждествено вярна в* $\mathbf{M}$*.*

✓ Доказателство. Нека $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са свободните променливи на $\exists \mathtt{x}\,(\varphi)$; тогава свободните променливи на φ са измежду $\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

Нека $\varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$ е скулемово усилване на $\exists \mathtt{x}\,(\varphi)$, което е тъждествено вярно в $\mathbf{M}$. За да докажем $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$, да изберем произволна оценка v в $\mathbf{M}$. Трябва да докажем $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[v]$. От $\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$ следва

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n][v]$$

Съгласно твърдение 3.3.17 това е еквивалентно на

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq [\![\tau]\!]^{\mathbf{M}} v, v(\mathtt{x}_1), v(\mathtt{x}_2), \ldots, v(\mathtt{x}_n)]$$

което съгласно дефиниция 3.3.2 ж) ни дава

$$\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq v(\mathtt{x}_1), v(\mathtt{x}_2), \ldots, v(\mathtt{x}_n)]$$

т.е.

$$\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[v]$$

■

✓ **3.7.22. Твърдение.** *Нека φs е скулемово усилване на $\exists \mathtt{x}\,(\varphi)$ и скулемовият символ не се среща никъде във формулата φ. Ако $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$, то съществува структура $\mathbf{K}$, която съвпада с $\mathbf{M}$ във всичко, освен може би при интерпретацията на скулемовия символ, такава, че $\mathbf{K} \vDash \varphi s$.*

Доказателство. Доказателството ще извършим по отделно в зависимост от това дали скулемовото усилване е от първи или втори вид.

(първи вид) Съгласно дефиниция 3.7.18 субституцията s има вида $\mathtt{x} \coloneqq \mathtt{c}$, където символът за константа $\mathtt{c}$ не се среща никъде във формулата φ, а формулата $\exists \mathtt{x}\,(\varphi)$ няма свободни променливи. Тъй като $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$, то съгласно дефиниция 3.3.2 ж) получаваме, че съществува такова $\mu \in |\mathbf{M}|$, че

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \mu]$$

Нека структурата $\mathbf{K}$ е идентична с $\mathbf{M}$ във всичко, освен в интерпретацията на символа $\mathtt{c}$ — нека $\mathtt{c}^{\mathbf{K}} = \mu$.

За да установим, че $\mathbf{K} \vDash \varphi[\mathtt{x} \coloneqq \mathtt{c}]$, да изберем произволна оценка v. Трябва да докажем

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \mathtt{c}][v]$$

Съгласно твърдение 3.3.17 това е еквивалентно на

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq [\![\mathtt{c}]\!]^{\mathbf{M}} v]$$

т.е. на

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \mu]$$

което вече видяхме, че е вярно.

(втори вид) Съгласно дефиниция 3.7.18 субституцията s има вида $\mathtt{x} \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, където функционалният символ $\mathtt{f}$ не се среща никъде във формулата φ, а $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са свободните променливи на $\exists \mathtt{x}\,(\varphi)$. Тъй като $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$, то за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$

$$\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu_1, \mu_2, \ldots, \mu_n]$$

откъдето съгласно дефиниция 3.3.2 ж) получаваме, че за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ съществува такова $\mu \in |\mathbf{M}|$, че

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu, \mu_1, \mu_2, \ldots, \mu_n] \tag{18}$$

Нека $f\colon |\mathbf{M}|^n \to |\mathbf{M}|$ е функция, която на произволно избрани $\mu_1, \mu_2, \ldots, \mu_n$ съпоставя така намереното μ.*

Нека структурата $\mathbf{K}$ е идентична с $\mathbf{M}$ във всичко, освен в интерпретацията на символа $\mathtt{f}$ — нека $\mathtt{f}^{\mathbf{K}} = f$.

За да установим, че $\mathbf{K} \vDash \varphi[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)]$, да изберем произволна оценка v. Трябва да докажем

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n), \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n][v]$$

Съгласно твърдение 3.3.17 това е еквивалентно на

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq [\![\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)]\!]^{\mathbf{M}} v, [\![\mathtt{x}_1]\!]^{\mathbf{M}} v, \ldots, [\![\mathtt{x}_n]\!]^{\mathbf{M}} v]$$

т.е. на

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq f(v(\mathtt{x}_1), \ldots, v(\mathtt{x}_n)), v(\mathtt{x}_1), \ldots, v(\mathtt{x}_n)]$$

Последното е истина предвид (18) и дефиницията на функцията f. ∎

Забележка: В доказателството на твърдение 3.7.22 за втория вид скулемово усилване използвахме т. н. *аксиома за избора*. Един начин да формулираме тази аксиома е следният:

> Ако за всяко $x \in X$ съществува $y \in Y$, за които е верен някакъв предикат $p(x, y)$, то тогава съществува такава функция f, че за всяко $x \in X$ е вярно $p(x, f(x))$.

Ако тук интерпретираме квантора „съществува“ конструктивно, тази аксиома е съвсем естествена, защото в този случай ще разполагаме с конкретен метод, посредством който по дадено x можем да получим нужното y. Когато интерпретираме квантора класически, не разполагаме с никакъв метод, посредством който по x можем да получим y. Затова аксиомата за избора дълго време е била една от най-оспорваните аксиоми на теория на множествата. Всъщност методът на скулемизацията, може да се обоснове и без да се използва аксиомата за избора, но доказателството става значително по-усложнено.

✓ **3.7.23. Дефиниция.** Множество от формули е *изпълнимо*, ако съществува структура, в която са тъждествено верни всички формули от множеството. Множество от формули е *неизпълнимо*, ако не е изпълнимо.

*На това място използваме т. н. *аксиома за избора*.

3.7.24. (скулемизация) Нека ни е дадено крайно множество от формули Γ в пренексна нормална форма и да разгледаме следния процес, който ще наречем *скулемизация*. Избираме произволна формула от Γ, която съдържа квантор. Ако формулата започва с квантор $\forall$, то отстраняваме този квантор. Съгласно твърдение 3.7.17 ще получим такова множество от формули Γ', че Γ е изпълнимо тогава и само тогава, когато Γ е изпълнимо. Ако пък избраната формула започва с квантор $\exists$, то да заменим тази формула с такова нейно скулемово усилване, че СКУЛЕМОВИЯТ ФУНКЦИОНАЛЕН СИМВОЛ ДА НЕ СЕ СРЕЩА В НИКОЯ ФОРМУЛА ОТ Γ.* Съгласно твърдение 3.7.21, ако Γ' е изпълнимо, то и Γ ще е изпълнимо. Ще докажем и обратното — ако Γ е изпълнимо, то и Γ' е изпълнимо. Да допуснем, че Γ е изпълнимо и нека $\mathbf{M}$ е някоя структура, в която са тъждествено верни формулите от Γ. Съгласно твърдение 3.7.22 или 3.7.22 съществува структура $\mathbf{K}$, в която е вярно скулемовото усилване на формулата, започваща с $\exists$. Останалите формули са идентични в Γ и Γ'. Те обаче не съдържат скулемовия символ, а структурите $\mathbf{M}$ и $\mathbf{K}$ съвпадат за всички символи, освен евентуално за скулемовия. Тъй като тези формули са тъждествено верни в $\mathbf{M}$, то те са тъждествено верни и в $\mathbf{K}$.

Тъй като на всяка стъпка от описания процес изчезва по един квантор, то след краен брой стъпки ще получим множество, в което всички формули са безквантворни. При това, така намереното множество не се различава от първоначалното по отношение на своята изпълнимост — ако първоначалното множество е изпълнимо, то и новополученото ще е неизпълнимо, и ако новополученото е изпълнимо, то значи и първоначалното множество е било изпълнимо.

И така, доказахме следната теорема:

✓ **3.7.25. Теорема.** *Нека Γ е крайно** множество от формули. Ако в сигнатурата има достатъчно символи, то ще можем да намерим такова крайно множество Γ' от безквантворни формули, че Γ да бъде изпълнимо тогава и само тогава, когато е изпълнимо Γ'.*

Доказателство. Да приведем формулите от Γ в пренексен вид и след това да приложим скулемизация (вж. 3.7.24). ∎

✓ **3.7.26. Дефиниция.** Казваме, че една формула е в *скулемова нормална форма*, ако:

*Разбира се, може да направим това само ако в сигнатурата има достатъчно символи.

**Теоремата е вярна и за безкрайно Γ, но тук няма да обосноваваме това.

– формулата е в пренексна нормална форма;

– формулата не съдържа нито един квантор $\exists$;

– формулата не съдържа свободни променливи.

✓ **3.7.27. Теорема за скулемизацията.** *Нека* Γ *е крайно*[*] *множество от формули. Ако в сигнатурата има достатъчно символи, то ще можем да намерим такова крайно множество* Γ' *от формули в скулемова нормална форма, че* Γ *да бъде изпълнимо тогава и само тогава, когато е изпълнимо* Γ'.

✓ Доказателство. Най-напред, използвайки теорема 3.7.25 можем да получим крайно множество Δ от безкванторни формули, което е изпълнимо тогава и само тогава, когато е изпълнимо Γ. Ако пред всяка от безкванторните формули в Δ сложим достатъчно квантори за всеобщност, ще получим множество Γ' в скулемова нормална форма. Освен това, съгласно твърдение 3.7.17, формулите от множеството Γ' са тъждествено верни точно в онези структури, в които са тъждествено верни и формулите от Δ. Следователно Δ е изпълнимо тогава и само тогава, когато е изпълнимо Γ'. ∎

3.7.28. Твърдение. *За всеки три формули* φ, ψ *и* χ*:*

а) $\vDash \varphi \vee (\psi \,\&\, \chi) \Leftrightarrow (\varphi \vee \psi) \,\&\, (\varphi \vee \chi)$

б) $\vDash (\psi \,\&\, \chi) \vee \varphi \Leftrightarrow (\psi \vee \varphi) \,\&\, (\chi \vee \varphi)$

Доказателство. Да изберем произволна структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, B е съждението $\mathbf{M} \vDash \psi[v]$ и C е съждението $\mathbf{M} \vDash \chi[v]$. Трябва да докажем, че са верни твърденията

> „Съждението (A или (B и C)) е вярно тогава и само тогава, когато е вярно ((A или B) и (A или C))“

и

> „Съждението ((B и C) или A) е вярно тогава и само тогава, когато е вярно ((B или A) и (C или A))“

И двете твърдения са очевидни. ∎

✓ **3.7.29. Дефиниция.**
Литерал означава атомарна формула или отрицание на атомарна формула.

[*] Теоремата е вярна и за безкрайно Γ, но тук няма да обосноваваме това.

б) *Елементарна дизюнкция* означава безкванторна формула, в която единствените логически операции са дизюнкция и отрицание и всяко отрицание се намира пред атомарна формула.

в) Една безкванторна формула е в *конюнктивна нормална форма*, ако представлява конюнкция от елементарни дизюнкции.*

✓ **3.7.30. Твърдение.** *За всяка безкванторна формула можем да намерим еквивалентна на нея (според класическата логика) формула, която е в конюнктивна нормална форма.*

✓ Доказателство. Нека φ е произволна безкванторна формула. Да я приведем в отрицателна нормална форма.** По този начин ще получим безкванторна формула, в която единствените логически операции са конюнкция, дизюнкция и отрицания и всяко отрицание се намира пред атомарна формула. Да прилагаме към така получената формула докато може замени от следния вид:

$$\varphi \vee (\psi \,\&\, \chi) \longmapsto (\varphi \vee \psi) \,\&\, (\varphi \vee \chi) \tag{19}$$

$$(\psi \,\&\, \chi) \vee \varphi \longmapsto (\psi \vee \varphi) \,\&\, (\chi \vee \varphi) \tag{20}$$

Съгласно твърдение 3.7.28 при замени от този вид ще получаваме еквивалентни формули. Освен това след всяка такава замяна формулата ще остава безкванторна, няма да се появят други операции, освен конюнкция, дизюнкция и отрицание и всяко отрицание ще си остане пред атомарната си формула. Може да забележим обаче, че ако след краен брой стъпки стигнем до формула, към която не можем да приложим

*По точно, можем да дадем следната индуктивна дефиниция на това какво значи *конюнктивна нормална форма*:

а) всяка елементарна дизюнкция е в конюнктивна нормална форма;

б) конюнкция на две формули в конюнктивна нормална форма е формула в конюнктивна нормална форма.

**Да припомним, че за целта е достатъчно да прилагаме докато може замени от следния вид:

$$\varphi \Leftrightarrow \psi \longmapsto (\varphi \Rightarrow \psi) \,\&\, (\psi \Rightarrow \varphi)$$

$$\varphi \Rightarrow \psi \longmapsto \neg\varphi \vee \psi$$

$$\neg\neg\varphi \longmapsto \varphi$$

$$\neg(\varphi \vee \psi) \longmapsto \neg\varphi \,\&\, \neg\psi$$

$$\neg(\psi \,\&\, \varphi) \longmapsto \neg\psi \vee \neg\varphi$$

замяна от горния вид, това ще означава, че сме получили формула в конюнктивна нормална форма. Това показва частичната коректност на алгоритъма за привеждане в конюнктивна нормална форма. Остава да видим защо алгоритъмът винаги свършва след краен брой стъпки.

(I начин) За произволна формула χ да наречем *дълбочина* на подформулата χ' на χ дължината на най-дългата редица от вида

$$\chi_1, \chi_2, \chi_3, \ldots, \chi_k$$

където $\chi_1 = \chi$, $\chi_k = \chi'$, $\chi_i \neq \chi_{i+1}$, χ_{i+1} е подформула на χ_i и формулите $\chi_1, \chi_2, \ldots, \chi_{k-1}$ не са от вида $\chi' \,\&\, \chi''$. Ако си мислим формулата като дърво, тогава дълбочината на една подформула е равна на разстоянието между корена на поддървото и корена на цялото дърво, като при това пропускаме възлите от дървото, в които стои конюнкция.

Може да се забележи, че ако формулата ψ се получава от φ посредством някоя от замените (19) и (20), то броят на подформулите от вида $\chi' \,\&\, \chi''$ не се променя, а дълбочината на някоя подформула от този вид намалява. Това не може да продължи до безкрайност.

(II начин) Да разгледаме следния начин, по който от безкванторна формула в отрицателна нормална форма получаваме аритметичен израз. Да заменим всеки литерал с числото 2. Да заменим всяка подформула от вида $\chi' \,\&\, \chi'$ с $x + y$, където x и y са изразите, получени съответно от χ' и χ''. Да заменим всяка подформула от вида $\chi' \vee \chi'$ с $x.y$, където x и y са изразите, получени съответно от χ' и χ''. След всяко преобразование на формулата от горния вид, стойността на съответния аритметичен израз няма да се промени, защото $x.(y + z) = x.y + x.z$ и $(y + z).x = y.x + z.x$. Същевременно след всяко такова преобразование се увеличава броят на числата 2 в така получения аритметичен израз. Това не може да продължи до безкрайност, защото за числа не по-малки от 2 е вярно $x + y \leqq x.y$ и значи стойността на всеки такъв аритметичен израз със сигурност е не по-малка от удвоения брой на числата 2 в него. ∎

3.7.31. Твърдение. *Ако в сигнатурата разполагаме с достатъчно неизползвани символи за константи и функционални символи,[*] то за всяко крайно множество[**] от формули можем да намерим такова*

[*] На практика това не е сериозно ограничение, защото при нужда винаги можем да заменим сигнатурата с нова сигнатура, в която има много нови символи.

[**] Това твърдение е вярно и за безкрайни множества, но тук няма да доказваме това.

крайно множество от елементарни дизюнкции, че (от гледна точка на класическата логика) първоначалното множество е изпълнимо тогава и само тогава, когато е изпълнимо множеството от елементарни дизюнкции.

Доказателство. Съгласно теорема 3.7.25 за всяко крайно множество от формули можем да намерим крайно множество от безкванторни формули, което е изпълнимо тогава и само тогава, когато е изпълнимо първоначалнорагато множество. Също така видяхме, че за всяка безкванторна формула можем да намерим еквивалентна на нея безкванторна формула в конюнктивна нормална форма.

Всяка формула от вида $\varphi \& \psi$ е тъждествено вярна в някоя структура $\mathbf{M}$ тогава и само тогава, когато формулата $\varphi \& \psi$ е вярна при всяка оценка в $\mathbf{M}$, а това е така тогава и само тогава, когато при всяка оценка в $\mathbf{M}$ са верни двете формули φ и ψ, и значи тогава и само тогава, когато тези две формули са тъждествено верни в $\mathbf{M}$. Това означава, че ако в така полученото множество заменяме докато може всяка формула от вида $\varphi \& \psi$ с двете формули φ и ψ, ще получаваме формули, които са изпълними тогава и само тогава, когато е изпълнимо първоначалното множеството от формули. Тъй като след всяка такава замяна броят на конюнкциите намалява, то след краен брой стъпки ще стигнем до множество от елементарни дизюнкции. ■

Твърдение 3.7.31 показва, че когато се интересуваме от изпълнимостта на множество от формули по отношение на класическата логика, може да считаме, че елементите на това множество имат сравнително прост вид — елементарни дизюнкции.

Глава 4

Логическо програмиране

4.1. „Разсъждаващи“ компютри

В знанията е силата!

Едуард Файгенбаум

Бази знания

Първите компютърни програми са използвали прости структури данни — числа, вектори и матрици, т.е. типове, заимствани от линейната алгебра. Първият компилатор на ФОРТРАН от 1957 г. вероятно е и първата компютърна програма, използваща сложни структури данни. Но макар и сложни, тези структури данни все още нямат собствен живот — програмата ги създава, използва ги, но след като приключи работата си, тези данни изчезват.

През 60-те години на XX век се появяват първите *бази данни*. При тях данните за пръв път получават съществуване, което е независимо от това на използващите ги програми. След като програмата си свърши работата, базата данни продължава съществуването си, очакваща ново стартиране на програмата. Често една база данни се използва не от една, а от няколко независими една от друга програми.

Едно характерно свойство на базите данни е тяхната статичност — въпреки че програмите могат да изтриват, добавят и изменят съществуващите данни, ако някоя програма не извърши някоя от тези опера-

ции, базата данни ще си стои без изменения. С други думи макар тук данните да са придобили независимо съществуване от това на програмите, всички изменения в данните се извършват под непосредствения контрол на използващите ги програми.

Първата компютърен програма, която можела да извежда нови знания, които не били част от първоначалната база, е Логическият теоретик, разработен от Хърбърт Саймън, Джон Клифърд Шо и Алън Нюъл през 1955. Само две години по-късно Хърбърт Саймън направил следното гръмко изявление:

> *Не искам да ви слисвам или ужасявам, ама най-простият начин да бъда кратък е като кажа, че на света вече има машини, които мислят, които се учат и които творят. При това способността им да правят тези неща има да расте бързо, тъй че в крайна сметка — в обозримото бъдеще — видовете задачи, с които те ще могат да работят, ще бъде съпоставими с тези, с които се е занимавал човешкият разум.*

Но не само авторите на Логическия теоретик били такива невероятни оптимисти. И останалите занимаващи се с изкуствен интелект не падали по-долу. Целите, които те си поставяли, били величествени, предвижданията за бъдещето — грандиозни, а постигнатото — незначително.*

Първият реален успех дошъл едва тогава, когато на изследователите им се наложило да решат не някоя впечатляваща задача с невъзможно решение, а практическа задача, поставена от възложител, който плащал и искал работещ софтуер. През 1965 НАСА планирала да изпрати автоматичен апарат на Марс, снабден със спектрометър за изследване на марсианската почва. По принцип въз основа на спектралните данни е възможно да се определят веществата, обаче не съществува точен метод как да се прави това. Химическите лаборатории обикновено използват метода на пробите и грешките, като се опитват да синтезират вещество, отговарящо на спектралните данни. Когато обаче не знаем абсолютно нищо за изследваното вещество, тогава има милиони вещества, които трябва да се пробват. За да избегне това, НАСА се обърнала

*Когато в областта на изкуствения интелект се появи нов метод, първите резултати обикновено са изключително обнадеждаващи. Например Логическият теоретик успял да докаже 38 от първите 52 теореми във втора глава на Principia Mathematica [21]. Доказателството на теорема 2.85 дори се оказало по-елегантно, от това на авторите Ръсел и Уайтхед. По това време Ръсел бил все още жив и Саймън успял да му покаже новото доказателство, на което Ръсел отвърнал с възхищение.

към изследователите в Станфордския университет с молба да видят дали не е възможно този анализ да се улесни с компютърна програма.

Така се появила първата *експертна система*, наречена ДЕНДРАЛ. Успехът на тази програма се дължал на това, че тя не използвала някакъв изтънчен логически метод за доказателства, ами просто притежавала знанията, систематизирани от хора-експерти, умеещи да правят спектрален анализ. От логическа гледна точка ДЕНДРАЛ е пределно проста система. Единственият вид разсъждения, които тя правела, са от вида „всички човеци са смъртни, а Сократ е човек, значи Сократ е смъртен“. Не било нужно нищо повече от Аристотелевата силогистика. Тъй като системата ДЕНДРАЛ наблягала на знанията, а не на някакви сложни логическите методи за извод, Едуард Файгенбаум, един от разработчиците на ДЕНДРАЛ, с основание възкликнал „В знанията е силата!“.

Най-съществената разлика между базите данни и *базите знания* е това, че при последните има възможност да се извеждат нови знания, които не са били част от първоначалната база. Един много важен практически проблеми, свързан с базите знания, е това какво компютърно представяне на знанията да се използва. То трябва да притежава следните свойства:

– да позволява ефективен извод на нови знания;

– да позволява представянето на всички необходими знания;

– да бъде удобен за използване от хора.

За съжаление е невъзможно да удовлетворим едновременно всяко едно от тези свойства. Колкото по-разнообразни видове знания могат да бъдат компютърно представени, толкова по-неефективно става извеждането на нови знания от вече наличните. Ако пък ограничим видовете знания, които могат да бъдат представени, тогава значително ще ограничим приложимостта на разработената система.

Нуждата представянето на знанията да бъде удобно за използване не само от компютри, но и от хора, още повече усложнява нещата. Когато използваме естествен език, например български, ние приемаме много неща за подразбиращи се. Когато хората разговарят, слушателите тълкуват не само прекия смисъл на казаното, но се опитват да възстановят и мисловния контекст, който е довел до съответното изказване. Например ако някой турист е бил в Охрид и като се върне каже „Белите кучета в Охрид имаха рунтави опашки“, всеки нормален човек* ще си направи извода, че не всички кучета в Охрид са има-

*Разбира се, някои математици, логици, програмисти и други подобни субекти не се включват в категорията „всички нормални хора“.

ли рунтави опашки. Наистина, ако всички кучета бяха имали рунтави опашки, не би имало смисъл да се казва, че само белите кучета са били с рунтави опашки. От гледна точка на логиката обаче, това е неправилен извод. От това че белите кучета са имали рунтави опашки не следва логически, че не всички кучета са с рунтави опашки! Напротив, един логически разсъждаващ естествоизпитател би казал, че щом белите кучета са били с рунтави опашки, а за останалите нямаме никакви данни, то значи въз основа на наличните данни най-правдоподобна е хипотезата, че и останалите кучета са били с рунтави опашки.

Компютрите не могат сами да добавят контекст към формулировките на съждения и затова знанията в базите знания трябва да се представят изчерпателно, без подразбиращи се неща и съгласно правилата на логиката, а не според обичайния начин, използван в ежедневието. За съжаление, поради начина, по който мислим, на хората ни е трудно да формулираме знанията си изчерпателно.[*]

Правила

Възможността да се извеждат нови знания, от вече наличните, е едно от най-важните свойства на базите знания. *Правилата* са представяне на данните, при което автоматичното извеждане на нови знания става максимално опростено. За представяне на знанията в първата експертна система ДЕНДРАЛ се използвали правила. И макар вече да има и алтернативни представяния на знанията, правилата си остават основният начин за представяне на знания в експертните системи.

Ето някои примерни правила за „мотоциклетна“ експертна система:[**]

1. Ако мотоциклетът при движение се тресе или придърпва, то да се заменят свещите.
2. Ако мотоциклетът при движение се тресе или придърпва и свещите са заменени, да се провери дали има свободен път за горивото от карбуратора до двигателя.

[*] Всъщност не е възможно човек да се научи да разсъждава логически правилно. Дори ако той е професор по математическа логика. Това се вижда например от статията [22] на Юдковски, вж. `https://intelligence.org/files/CognitiveBiases.pdf`. Но въпреки че никой човек не може да разсъждава логически правилно във всичко, на изпита по логическо програмиране от студентите се очаква да направят точно това.

[**] Тези правила са адаптация на правила от [14].

3. Ако мотоциклетът при движение се тресе или придърпва, свещите са заменени и няма запушване или теч на гориво между карбуратора и двигателя, да се провери въздуховодът до карбуратора.
4. Ако мотоциклетът при движение се тресе или придърпва, свещите са заменени, няма запушване или теч на гориво между карбуратора и двигателя и въздуховодът до карбуратора е наред, да се отиде при техник.
5. Ако е натиснат съединителя и той не задейства, да се регулира пружината на съединителя.
6. Ако пружината на съединителя е регулирана, но при натискане той не задейства, то да се потегли на втора скорост, след което едновременно се натиска съединителя, дава се газ и се натиска спирачката.
7. Ако при натискане на съединителя, той не задейства, пружината му е регулирана и опитът да се освободи с едновременно прилагане на спирачка и газ е неуспешен, то да се отиде при техник.

Виждаме, че тези правила приличат на логически съждения от следния вид:

$$\text{АКО } \varphi_1 \text{ И } \varphi_2 \text{ И } \ldots \text{ И } \varphi_n, \text{ ТО } \psi$$

Тук $\varphi_1, \varphi_2, \ldots, \varphi_n$ се наричат *предпоставки* или *антецеденти* на правилото, а ψ — негово *заключение*, *консеквент*. В математическата логика най-често се използва вертикален запис на правилата. Той изглежда така:

$$\frac{\varphi_1 \quad \varphi_2 \quad \ldots \quad \varphi_n}{\psi}$$

Когато дадено правило няма предпоставки

$$\frac{}{\psi}$$

такова правило ни казва, че ψ е винаги вярно. В математическата логика правилата без предпоставки се наричат *аксиоми*, а в логическото програмиране — *факти*.

Посочените по-горе „мотоциклетни“ правила взаимодействат едно с друго по прост и предвидим начин. Не е трудно да се направи програма на традиционен език за програмиране, която посредством няколко вложени оператора `if ... then ... else ...` изпълнява тези правила. Обикновено обаче правилата на експерните системи се задействат по непредвидим начин. Да разгледаме например следното правило от експертна система за диагностика на бактериални инфекции:

Ако оцветяването на организма е грамотрицателно и формата на организма е пръчица и по отношение на аеробността организмът е аеробен, то тогава има голяма вероятност (0,8) организмът да е вид ентеробатерия.

Само по себе си това правило не ни подсказва как трябва да го използваме. Какъв смисъл има да знаем, че дадена бактерия е ентеробактерия? И каква е ползата, ако дори не сме сигурни, че това е ентеробактерия, а само сме приели, че има голяма вероятност това да е така? Отговорът на тези въпроси зависи по сложен начин от останалите правила. Това е причината обикновено правилата в експертната система да не се превръщат директно в програмен код, а да се използват посредством специална дедуктивна машина (inference engine) — алгоритъм, чиято функция е да извежда автоматично нови знания от вече съществуващите.

Правилата представляват основният метод за представяне на данни в съвременните експертни системи. В някои случаи обаче е по-удачно да се използват алтернативни методи:

Фреймове. Напомнят класовете и обектите при обектноториентираното програмиране, но всъщност са предложени от Мински още през 1974 г., т.е. преди появата на обектноориентираните езици. Фреймовете позволяват да се разсъждава по аналогия. За да опишем даден обект, ние наследяваме свойствата от вече съществуващи класове и казваме само какви са разликите. Нека например фреймът „птица“ описва свойствата на птиците, едно от които е това че те летят. В такъв случай можем да посочим, че фреймът „пингвин“ е подклас на фрейма „птица“, при което пингвините ще наследят всички типични свойства на птиците, а за нас остава само да опишем специфичните особености, например това че пингвините не летят. Фреймовете са удобни при разработката на симулационни експертни системи.

Размита логика. Специфичното при този вид експертни системи е това, че те използват размитата логика за да дадат „дефиниция“ на думи като „много“, „малко“, „топло“, „хладко“, „студено“, „бързо“, „бавно“ и т.н.* По този начин става по-лесно знанията, получени от експертите, да бъдат преведени на „компютърен език“. Този тип експертни системи не са надеждни, когато трябва да генерираме отговор на въпрос от типа да/не, например дали е целесъобраз-

*Понякога за този тип експертни системи неправилно се твърди, че специфичното при тях е това, че те могат да работят с несигурни данни. Това не е така, защото дори някои от най-старите експертни системи умеят да работят с несигурни данни.

но даден пациент да се оперира. За сметка на това обаче те се спрявят доста добре когато отговорът на въпроса не е дискретен, например с каква скорост е най-добре да се движи метровлакът. Знанията се представят във вид на размити правила. Една особеност на експертните системи, основаващи се на размита логика, е това, че те използват правилата по схематичен и донякъде предвидим начин, а не произволно, както при експертните системи с обикновени правила.

Невронни мрежи. Има някои неща, които правим лесно и без да се замисляме, но всъщност са много трудни. Как например обработваме звуковия сигнал на говор, за да го „превърнем“ в редица от думи? Как разбираме дали дадено лице е мъжко или женско? Как пазим равновесие когато вдигаме предмет с неизвестно тегло или когато ходим по хлъзгав терен? Тъй като не знаем отговорите на тези въпроси, не можем да ги опишем посредством правила. Налага се компютърът сам „да се научи“ и невронните мрежи са един от успешните методи да се прави това. В последно време, използвайки трениране посредством т.н. deep learning, се оказва възможно да обучим невронни мрежи, решаващи много сложни задачи.

Права и обратна изводимост

Има два основни начина за компютърно използване на правила. При първия начин, т.н. *права изводимост*, ако в базата знания вече се съдържат предпоставките на дадено правило като установени знания, към нея ще бъде добавено и заключението на правилото. Например, ако имаме правило ψ :- $\varphi_1, \varphi_2, \ldots, \varphi_n$ и предпоставките $\varphi_1, \varphi_2, \ldots, \varphi_n$ са вече известни знания, към базата знания ще бъде добавено и заключението ψ. Когато се използва права изводимост системата се интересува какви знания има до момента и въз основа на тях извежда нови знания. При правата изводимост обаче системата не се интересува каква е крайната цел и затова често генерира излишно много ненужни знания. Това е и основният недостатък на правата изводимост.

При *обратната изводимост* правилата се използват „наопаки“. Тук системата обръща основно внимание на това каква е целта, която искаме да постигнем, и чрез кои правила тя може да се получи. След като открие тези правила, с тяхна помощ се пораждат нови подцели. Тези подцели пораждат нови подцели и т.н. дотогава, докато открием начин да се постигне първоначалната цел. Недостатък на обратната изводимост е това, че системата не се интересува какво всъщност е известно,

така че ще пробва много невъзможни начини, за постигане на желаната цел. Обратната изводимост не е много подходяща и за управляващи системи, реагиращи на нововъзникнали събития (например спиране подаването на гориво към двигателя, ако данните, постъпили от датчиците, подсказват, че има пожар).

Ще илюстрираме правата и обратната изводимост с пример за химичен синтез.[*] Да допуснем, че разполагаме с неограничено количество въглерод (C), кислород (O_2), водород (H_2) и магнезиев оксид (MgO) и искаме да получим въглеродна киселина (H_2CO_3). За целта можем да използваме следните химични реакции:

$$\begin{gathered} MgO + H_2 \longrightarrow Mg + H_2O \\ C + O_2 \longrightarrow CO_2 \\ CO_2 + H_2O \longrightarrow H_2CO_3 \end{gathered}$$

Наличието на въглерод, кислород, водород и магнезиев оксид можем да представим посредством следните аксиоми:

$$\overline{C} \quad (21) \qquad \overline{O_2} \quad (22) \qquad \overline{H_2} \quad (23) \qquad \overline{MgO} \quad (24)$$

Трите позволени химични реакции пък се свеждат до следните четири правила:

$$\frac{MgO \quad H_2}{Mg} \quad (25) \qquad \frac{MgO \quad H_2}{H_2O} \quad (26)$$

$$\frac{C \quad O_2}{CO_2} \quad (27) \qquad \frac{CO_2 \quad H_2O}{H_2CO_3} \quad (28)$$

Целта е да получим въглеродна киселина

$$H_2CO_3$$

Да видим как всичко това работи при използване на права изводимост. Да означим с Δ_i множеството на нещата, които са известни на стъпка i. В началото все още нищо не е известно (т.е. базата знания е празна) и затова

$$\Delta_0 = \varnothing$$

Единствените правила, които могат да се задействат[**] при положение, че все още нищо не е известно, са правилата без предпоставки, т.е. (21)–(24). След като тези правила се задействат

$$\Delta_1 = \{C, O_2, H_2, MgO\}$$

[*] Примерът е взет от [7, раздел 2.6].

[**] На английски технически жаргон се използва „fire“ вместо „задействат“.

дължина на търсения извод	***количество безполезни начини за търсене на извода***	***изводимост, която върши работа***
къс	малко	права и обратна
къс	много	права
дълъг	малко	обратна
дълъг	много	никоя

Таблица 4. Приложимост на правата и обратната изводимост

При новото състояние на базата знания правилата без предпоставки отново могат да се задействат, но няма да ни дадат нищо ново. Освен тях обаче вече може да се задействат и правила (25)–(27). С тяхна помощ базата знания става

$$\Delta_2 = \{\mathrm{C}, \mathrm{O}_2, \mathrm{H}_2, \mathrm{MgO}, \mathrm{Mg}, \mathrm{H}_2\mathrm{O}, \mathrm{CO}_2\}$$

При новото съдържание на базата знания правила (21)–(27) отново могат да се задействат, но няма да ни дадат нищо ново. Освен тях обаче може да се задейства и правило (28), посредством която и получаваме желаната цел $\mathrm{H}_2\mathrm{CO}_3$.

Да видим сега как можем да получим $\mathrm{H}_2\mathrm{CO}_3$, използвайки обратна изводимост.

Единственото правило, от което можем да получим $\mathrm{H}_2\mathrm{CO}_3$ е (28). Правило (28) ни дава две подцели — CO_2 и $\mathrm{H}_2\mathrm{O}$.

Единственото правило, от което можем да получим първата подцел, т.е. CO_2, е (27). От правило (27) получаваме две подподцели — C и O_2, които получаваме веднага от правилата без предпоставки.

Единственото правило, от която можем да получим втората подцел, т.е. $\mathrm{H}_2\mathrm{O}$, е (26). От правило (26) получаваме две подподцели — MgO и H_2, които получаваме веднага от правилата без предпоставки.

При този пример обратната изводимост се оказа по-ефективна, защото изобщо не ни се наложи да прилагаме правило (25). При други примери обаче правата изводимост може да се окаже по-ефективна от обратната. В таблица 4 може да се видят някои препоръки кога кой тип изводимост да се използва.

Да забележим, че и при правата, и при обратната изводимост открихме по същество един и същ начин за получаване на въглеродна киселина от въглерод, кислород, водород и магнезиев оксид. Логиците

изобразяват така намерения извод, използвайки дървоподобна* структура, изглеждаща по следния начин:

$$\dfrac{\dfrac{\mathrm{C} \qquad \mathrm{O_2}}{\mathrm{CO_2}} \qquad \dfrac{\mathrm{MgO} \qquad \mathrm{H_2}}{\mathrm{H_2O}}}{\mathrm{H_2CO_3}}$$

Всяка от хоризонталните черти в този запис отговаря на прилагането на някое от правилата.

***Задача* 46:** Дадени са правилата

$$\frac{}{\mathtt{p}} \qquad \frac{\mathtt{p} \quad \mathtt{p}}{\mathtt{q}} \qquad \frac{\mathtt{r}}{\mathtt{q}} \qquad \frac{\mathtt{s} \quad \mathtt{q}}{\mathtt{r}} \qquad \frac{\mathtt{q} \quad \mathtt{p}}{\mathtt{s}}$$

Използвайки обратна изводимост, изведете `r`.

Представане на правила посредством предикатни формули

За да можем да се възползваме наготово от всички неща, които сме направили за предикатната логика, е удобно да представяме правилата като специален вид формули, наречени хорнови клаузи.

✓ **4.1.1. Дефиниция.** Нека $n \geqq 0$ и формулите $\varphi_1, \varphi_2, \ldots, \varphi_n, \psi$ са атомарни. *Хорнови клаузи* ще наричаме формулите от следните два вида:

– ако $n = 0$, то атомарната формула ψ е хорнова клауза;

– ако $n \geqq 1$, то формулата $\varphi_1 \,\&\, \varphi_2 \,\&\, \cdots \,\&\, \varphi_n \Rightarrow \psi$ е хорнова клауза.

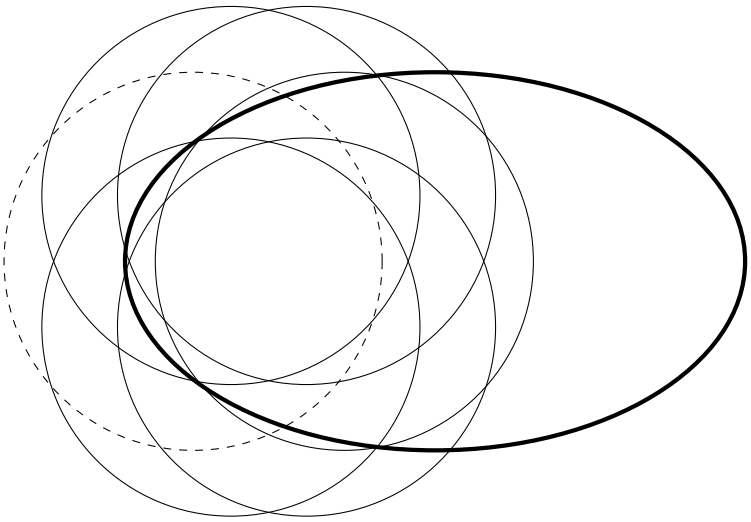
Атомарните формули $\varphi_1, \varphi_2, \ldots, \varphi_n$ се наричат *предпоставки* на клаузата, а атомарната формулата ψ — нейно *заключение*. За горните клаузи (включително когато $n = 0$) ще използваме следния запис:

$$\psi \,\texttt{:-}\, \varphi_1, \varphi_2, \ldots, \varphi_n$$

Клаузите без предпоставки (т. е. когато $n = 0$) се наричат *факти*. За краткост ще изпускаме прилагателното „хорнова“ и ще казваме само *клауза*.

***Забележка*:** Въпреки че на пролог фактите се записват без използването на знака `:-`, за да е ясно, че става въпрос за клауза, тук ще пишем ψ `:-` дори когато клаузата е без предпоставки. В контекста на пролог думата „правило“ означава не това, което бе казано по-горе. В езика пролог „правило“ означава хорнова клауза с поне една предпоставка. Следователно ако използваме терминологията на пролог, може да кажем, че клаузите са два вида: факти и правила.

*Логиците са единствените математици, при които дърветата растат в правилната посока, т.е. нагоре, а не надолу.



Фиг. 13. Диаграма на Вен за вярна клауза

4.1.2. Следствие. *Клаузата* ψ :- $\varphi_1, \varphi_2, \ldots, \varphi_n$ *е тъждествено вярна в структура* $\mathbf{M}$ *тогава и само тогава, когато при всяка оценка в* $\mathbf{M}$*, при която предпоставките* $\varphi_1, \varphi_2, \ldots, \varphi_n$ *са верни, заключението* ψ *също е вярно.*

Доказателство. Уверете се сами, че това наистина е така, използвайки дефиницията за тъждествена вярност (3.4.1 а)) и дефиницията за формула вярна в структура при оценка (3.3.2). Убедете се, че твърдението е вярно независимо дали $n = 0$ или $n \geqq 1$.

Обърнете внимание, че в общия случай за всяка предпоставка може да има оценки, при които тя е вярна, и оценки, при които тя не е вярна. Също и за заключението може да има оценки, при които то е вярно, и оценки, при които то не е вярно. А за да бъде една клауза тъждествено вярна в структура, ние искаме следното — при онези оценки, при които се окаже, че всички предпоставки са верни, заключението също да бъде вярно. Ако за някоя оценка дори една от предпоставките се окаже невярна, за такава оценка клаузата не казва нищо. Това е илюстрирано във фиг. 13. Всяка от окръжностите изобразява множеството от оценки, при които някоя от предпоставките е вярна. Елипсата с дебела линия пък е множеството от оценки, при които заключението е вярно. Вътрешната бяла област са оценките, при които всяка от предпоставките се оказва вярна и тази бяла област се включва изцяло в удебелената елипса. Забележете, че ако обръщаме внимание не на всички предпоставки, а само на една (напр. на щрихованата окръжност),

тогава в общия случай няма да открием никаква зависимост между нея и заключението. Зависимост съществува само когато вземем предвид всички предпоставки едновременно. ■

✓ **4.1.3. Дефиниция.** Нека $n \geqq 0$ и формулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ са атомарни. *Запитвания* се наричат формулите от следните два вида:

- когато $n = 0$, то формулата $\top$ е запитване (да припомним, че $\top = \neg\bot$ е винаги вярна формула);
- ако $n \geqq 1$, то формулата $\varphi_1 \,\&\, \varphi_2 \,\&\, \cdots \,\&\, \varphi_n$ е запитване

За запитванията (включително когато $n = 0$) ще използваме следния запис:

$$\texttt{?-}\ \varphi_1, \varphi_2, \ldots, \varphi_n$$

Запитването, което не съдържа нито една атомарна формула (т. е. когато $n = 0$), се нарича *празно запитване*.

4.1.4. Следствие. *а) Запитването* $\texttt{?-}\ \varphi_1, \varphi_2, \ldots, \varphi_n$ *е изпълнимо в структура* $\mathbf{M}$ *тогава и само тогава, когато може да се намери оценка в* $\mathbf{M}$*, при която формулите* $\varphi_1, \varphi_2, \ldots, \varphi_n$ *са верни.*

б) Празното запитване `?-` *е вярно във всяка структура.*

4.2. Права изводимост с частни случаи

Понятие за изводимост с частни случаи

Предпоставките и заключенията от правилата в примера за химичен синтез, който използвахме, за да се запознаем с правата и обратната изводимост (вж. стр. 215), не съдържаха променливи. Това, оказва се, спестява невероятно много усложнения. В този раздел ще разгледаме по-формално изводимостта с без променливи. Разбира се, вместо изрази като H_2O и NaCl ще използваме атомарни формули.

Нека например универсумът, с който работим, се състои от естествените числа. Да разгледаме клаузата:

$$\texttt{p(x, x, y) :-}$$

Тъй като универсум са естествените числа, то тази клауза е еквивалентна на следната безкрайна съвкупност от частни случаи без променливи:

$\mathtt{p}(0,0,0)$:-	$\mathtt{p}(0,0,1)$:-	$\mathtt{p}(0,0,2)$:-	$\mathtt{p}(0,0,3)$:-	$\mathtt{p}(0,0,4)$:-	...
$\mathtt{p}(1,1,0)$:-	$\mathtt{p}(1,1,1)$:-	$\mathtt{p}(1,1,2)$:-	$\mathtt{p}(1,1,3)$:-	$\mathtt{p}(1,1,4)$:-	...
$\mathtt{p}(2,2,0)$:-	$\mathtt{p}(2,2,1)$:-	$\mathtt{p}(2,2,2)$:-	$\mathtt{p}(2,2,3)$:-	$\mathtt{p}(2,2,4)$:-	...
$\mathtt{p}(3,3,0)$:-	$\mathtt{p}(3,3,1)$:-	$\mathtt{p}(3,3,2)$:-	$\mathtt{p}(3,3,3)$:-	$\mathtt{p}(3,3,4)$:-	...
$\mathtt{p}(4,4,0)$:-	$\mathtt{p}(4,4,1)$:-	$\mathtt{p}(4,4,2)$:-	$\mathtt{p}(4,4,3)$:-	$\mathtt{p}(4,4,4)$:-	...
...	...	...	...	...	...

Да забележим, че тези частни случаи всъщност не са клаузи. Та нали клаузите бяха специален вид формули, а формулите са редици от символи! Естествените числа обаче не са символи, следователно горните „клаузи“ не са нито формули, нито клаузи.

Това не е само формално заяждане. Да си представим например че универсум бяха реалните числа. Ами колкото и да сме хитри, няма как за всяко реално число да му измислим някакъв запис и следователно няма как да напишем частни случаи от горния вид за всяко реално число. Всъщност някои математици биха казали, че дори не за всяко естествено число можем да напишем символ. Числа като $10^{10^{10^{10}}}$ са толкова големи, че допускането, че можем измислим различни записи за естествените числа между 0 и $10^{10^{10^{10}}}$ е идеализация, която не съответства по никакъв начин на реалността. Според физиците в цялата вселена има не повече от 10^{100} елементарни частици (електрони, протони, неутрони, фотони, неутрино и др.), обаче числото $10^{10^{10^{10}}}$ е толкова по-голямо от този брой, че дори не можем да си представим колко точно по-голямо е то.* Понякога такива големи числа се наричат „астрономически“, но това е неправилно, защото и най-големите числа, с които боравят астрономите, са направо нищожни в сравнение с числа като $10^{10^{10^{10}}}$.

Има ли тогава смисъл да си губим времето с изводимост, използваща частни случаи като

$$\mathtt{p}(0,0,0), \mathtt{p}(0,0,1), \mathtt{p}(0,0,2) \text{ и т.н.?}$$

*Има математици, наречени ултрафинитисти, според които числото $10^{10^{10^{10}}}$ не съществува. Александър Сергеевич Есенин-Волпин е един от тях. Веднъж на една негова лекция Харви Фридман решил да разбере кое е най-малкото естествено число n, за което не съществува 2^n. Той попитал „Съществува ли 2^1?“. Есенин-Волпин отговорил „да“. После Фридман попитал „Ами 2^2?“. Пак бил получен отговор „да“, но след кратко забавяне. На въпроса за 2^3 Есенин-Волпин се замислил по-дълго, но все пак отговорил „да“. Това продължило с още няколко въпроса, но скоро станало ясно, че ако аудиторията иска да получи отговор „да“ че съществува 2^{100}, ще трябва да чака време 2^{100}, което няма как да стане.

Та нали една такава изводимост би била най-често невъзможна за реализация, а дори и да може в някои случаи да се реализира, то тя би се оказала изключително неефективна!

Въпреки горните притеснения, в много от учебниците по логика от край време са разглеждани изводимости с частни случаи. Това сигурно ще да е така, защото на някакви си математици им е интересна изводимост, от която не очакваме нищо смислено, и после карат и другите да я учат. Какъв абсурд!

При обратната изводимост в следващия раздел ще видим как тя може да бъде реализирана ефективно без да се използват частни случаи. По подобен начин и правата изводимост може да се реализира без да се използват частни случаи. Но въпреки това, като че ли нито една съвременна реализация на правата изводимост не прави това! Всички те реализират правата изводимост, използвайки частни случаи. Това е така, защото през 1974 г. Чарлз Форджи публикувал алгоритъмът *Peme*,[*] при което станало ясно, че всъщност има начин да реализираме ефективно правата изводимост с частни случаи. И така, оказва се, че това което отначало ни се стори невъзможно за реализация или най-малкото неефективно, всъщност е по-ефективно от алтернативите.

От тук можем да си извлечем най-малкото две поуки. Първата поука е това, че не винаги привидно невъзможните за програмиране неща са наистина невъзможни. Затова е добре да сме запознати от една страна с най-важните алгоритми, измислени от „умните хора“, а от друга страна с основните резултати в теория на изчислимостта, отнасящи се за неразрешими задачи и изчислителна сложност.

Втората поука е това, че няма теоретична част на математиката, за която да кажем, че не може да има приложения. Има много примери за неща, с които математиците са се занимавали, защото им харесва, без да ги е грижа дали тези неща имат практически приложения. Но въпреки това тези неща са намирали практически приложения, и то по

[*]Българският правопис изисква да записваме чуждите имена с кирилски букви, следвайки произношението в оригиналния език. Това създава проблеми при тази дума. На английски тя се изписва Rete, но всеки я произнася както си иска — рейти, рийти, рийт, ретей и др. По принцип в такива случаи трябва да се поинтересуваме как е измислена думата и от там да получим правилното произношение на английски, а от тук и на български. Да, но Чарлз Форджи е взел тази дума едновременно и от латински, и от италиански език. Латинската дума rete се произнася „рете“ и означава 'ловджийска мрежа'. Италианската дума rete се произнася „рите“ и също означава мрежа, но не задължително ловджийска. Тук сме превели думата Rete като Рете, използвайки произношението на латински език. Предимството на този превод е това, че при него записът на думата съвпада с транслитерацията на английската дума Rete с кирилски букви.

най-неочаквани начини. Невероятно е, но е факт, че по цял свят университетските преподаватели по теоретични математически дисциплини са запознати само откъслечно с приложенията на това, което преподават. Студентите сами трябва да откриват каква е практическата полза от преподаваните неща.*

Приложенията на теоретичната математика се откриват със закъснение, но са многобройни. Приложенията на приложната математика пък са ясни още от самото начало, но те често си остават такива, каквито са си били, т.е. нови приложения не се откриват. Трудно е да обясним защо това се случва. Вероятно причината трябва да се търси в това, което кара теоретичните математици да считат дадено нещо за „интересно“, „красиво“ или „смислено“. Няма да намерим математици, които да водят следния разговор:

— Много ми е интересно каква е трихиляди петстотин шестдесет и осмата цифра на числото π.

— О, така ли! Аз пък съм особено впечатлен от четири хиляди седемстотин и дванадесетата цифра.

— Абе най-смислена е единадесет хилядната цифра!

Макар тези три неща да се отнасят за математически резултати, тези резултати не са нито интересни, нито блестят с някаква красота, нито пък в тях виждаме особен смисъл.**

„Интересно“ е това, което отговаря на въпроси, които са възбудили любопитството на математика. А любопитството на математика се възбужда от очакването да открие нещо „красиво“ или „смислено“.

„Красив“ е математически резултат, който възбужда естетическа наслада, съчетавайки проста форма със сложно съдържание. Използвайки такива разултати математикът се надява, че с малки усилия ще може да отговаря на „интересни“ въпроси и да постига „смислени“ цели.

„Смислено“ е това, което ни помага да постигаме смислени цели. А „смислена“ цел за теоретичния математик е да се получи нещо „красиво“ или да се отговори на „интересен“ въпрос.

Безопасни правила

Да видим сега как можем да реализираме ефективно правата изводимост с частни случаи. Да си припомним, че при нея на всяка стъпка

*Не ми е известна книга по логика, в която след като е разказано за изводимостта с частни случаи, да е споменат и алгоритъмът Рете.

**Колцина след като прочетат горния диалог ще ги е грижа да проверят какви всъщност са тези цифри на числото π? Кой знае, може пък те да се окажат интересни?

използваме правилата, за да генерираме все по-големи множества

$$\Delta_0 \subseteq \Delta_1 \subseteq \Delta_2 \subseteq \Delta_3 \subseteq \ldots$$

от доказани факти. Нека например универсумът, с който работим, отново са естествените числа, а сред дадените правила е и следното:

$$\texttt{по-малко}(\texttt{x}, \texttt{y}) \texttt{ :- } \texttt{по-малко}(\texttt{x}, \texttt{z}), \texttt{по-малко}(\texttt{z}, \texttt{y}) \tag{29}$$

Нека на стъпка i сред фактите, които сме доказали, са и следните два:

$$\texttt{по-малко}(2, 3) \tag{30}$$
$$\texttt{по-малко}(3, 4) \tag{31}$$

Тъй като един от безброй многото частни случаи на правило (29) е и следният

$$\texttt{по-малко}(2, 4) \texttt{ :- } \texttt{по-малко}(2, 3), \texttt{по-малко}(3, 4) \tag{32}$$

то използвайки този частен случай, от (30) и (31) на стъпка $i + 1$ получаваме

$$\texttt{по-малко}(2, 4) \tag{33}$$

Да забележим, че от дадените неща на стъпка $i+2$ вече няма да може да се получи нищо ново. Макар правило (29) да има безброй много частни случаи, нито един от тях не може да бъде приложен към фактите (30), (31) и (33), така че да получим нов факт. Така правим следното важно наблюдение:

> *Не е нужно да генерираме всичките безброй много частни случаи на правилата, а само онези от тях, които могат да се приложат към вече доказаните факти.*

Значи ако на всяка стъпка доказаните факти са краен брой, то макар правилата да имат безброй много частни случаи, това не е пречка за компютърната реализация на правата изводимост.

А може ли да гарантираме, че на всяка стъпка доказаните факти ще бъдат краен брой? Изобщо възможно ли е ако на стъпка i имаме краен брой доказани факти, от тях на стъпка $i + 1$ да получим безброй много?

Оказва се, че това е възможно. Да разгледаме например правилото

$$\texttt{по-малко}(\texttt{x} + \texttt{z}, \texttt{y} + \texttt{z}) \texttt{ :- } \texttt{по-малко}(\texttt{x}, \texttt{y}) \tag{34}$$

Някои негови частни случаи са например следните:

$$\texttt{по-малко}(0,1)\texttt{ :- по-малко}(0,1)$$
$$\texttt{по-малко}(1,2)\texttt{ :- по-малко}(0,1)$$
$$\texttt{по-малко}(2,3)\texttt{ :- по-малко}(0,1)$$
$$\texttt{по-малко}(3,4)\texttt{ :- по-малко}(0,1)$$
$$\texttt{по-малко}(4,5)\texttt{ :- по-малко}(0,1)$$
$$\texttt{по-малко}(5,6)\texttt{ :- по-малко}(0,1)$$
$$\ldots\ldots\ldots$$

Вижда се, че дори ако единственият наличен факт беше $\texttt{по-малко}(0,1)$, то от него и изредените по-горе частни случаи бихме получили безброй много следствия $\texttt{по-малко}(1,2)$, $\texttt{по-малко}(2,3)$, $\texttt{по-малко}(3,4)$ и т.н.

Какво е по-различното в правило (34), поради което с това правило от краен брой вече известни неща може да изведем безброй много нови следствия? Защо безброй много негови частни случаи могат да се приложат към единствения факт $\texttt{по-малко}(0,1)$? Отговорът на тези въпроси се крие в променливата `z` — тя се среща единствено в заключението на правилото (т.е. от лявата страна), но не и в предпоставките (т.е. от дясната страна). Когато напаснем предпоставките на дадено правило към вече известни факти, то всички променливи, срещащи се в тези предпоставки, получават някакви конкретни стойности. Когато обаче някоя променлива, подобно на горното `z`, се среща само в заключението, тя не получава конкретна стойност и затова правилото може да се използва при произволни нейни стойности. Това ни мотивира да дадем следната дефиниция:

4.2.1. Дефиниция. Правило или клауза

$$\frac{\psi_1 \quad \psi_2 \quad \ldots \quad \psi_n}{\varphi} \qquad\qquad \varphi\texttt{ :- }\psi_1, \psi_2, \ldots, \psi_n$$

са *безопасни*, ако всяка променлива, която се среща в заключението φ, се среща също и някъде сред предпоставките $\psi_1, \psi_2, \ldots, \psi_n$.

Вижда се, че клауза (29) е безопасна, докато клауза (34) не е. Алгоритъмът *Рете* не е единственият, при който е нужно всички правила да бъдат безопасни. Езикът дейталог за заявки към бази данни не използва алгоритъма Рете, но и при него трябва да се погрижим всички правила да бъдат безопасни. За щастие, това обикновено не създава проблеми.

Когато всички правила са безопасни, тогава от краен брой факти с помощта на краен брой правила могат да се изведат само краен брой

нови факти. Това е така дори когато универсумът, с който работим, е неизброим, например множеството на реалните числа.

Но за да може да реализираме ефективно правата изводимост с частни случаи, това далеч не е единственото, за което трябва да се погрижим. Например какво трябва да се направи когато в множеството Δ_i се добави нов факт? Да проверяваме какви нови неща могат да се получат с всяко от наличните правила, едно по едно? Ясно е, че макар това да може да се реализира алгоритмично, ще получим изключително неефективна процедура, защото всеки път ще трябва да напасваме всяко от правилата с всяка комбинация от доказаните факти.

За да реши този проблем, алгоритъмът Рете използва специални структури данни, с помощта на които когато се появи нов факт, може много бързо да се види какви следствия може да получим с негова помощ. Алгоритъмът Рете е така направен, че без значение колко правила има при всеки нов факт можем да установим за константно време какви нови следствия може да се получат. За съжаление, за да постигне този забележителен резултат, алгоритъмът Рете използва твърде много оперативна памет — толкова много, че най-често това е ограничаващият фактор.

Чарлз Форджи не се ограничил само с изобретяването на алгоритъма Рете, но освен това го и реализирал в езика за програмиране ОПС5. Този език и до днес се използва в някои университетски курсове за експертни системи и бизнес правила. Сред по-сериозните реализации на алгоритъма Рете, които са свободен софтуер, трябва да посочим обвивката за експертни системи КЛИПС, разработена от НАСА.[*]

Забележка: По-нататък няма да предполагаме, че клаузите са безопасни, и ще развиваме теорията в общия случай — за произволни клаузи.

Обогатяване на структура

Когато пишем една компютърна програма, в нея ние дефинираме различни неща. Видът на тези неща зависи от използвания език за програмиране. Например в един функционален език дефинираме функции, в един логически език — предикати, в обектноориентираните езици се дефинират класове[**] и т.н.

[*] `www.clipsrules.net`

[**] От математическа гледна точка, когато не се използва наследяване, класовете може да се мислят като наредена двойка от многосортна сигнатура и структура за тази сигнатура. Сигнатурата се нарича *интерфейс* (джава, си++) или *протокол*

Някои от нещата, които се използват в нашата програма, са дефинирани не в самата нея, а се вземат наготово или от някоя програмна библиотека, или пък са вградени в самия език за програмиране. Следователно нещата, които се дефинират в нашата програма, разширяват съвкупност от вече съществуващи неща, които не е нужно ние да дефинираме. За такъв тип разширения в математическата логика се използва терминът „обогатяване“.

Да дефинираме формално какво означава обогатяване на сигнатура и обогатяване на структура:

4.2.2. Дефиниция. а) Сигнатурата $\mathfrak{sig}_1$ е *обогатяване* на $\mathfrak{sig}_2$, ако всеки символ на $\mathfrak{sig}_2$ е символ и на $\mathfrak{sig}_1$.

б) Структурата $\mathbf{M}_1$ е *обогатяване* на структурата $\mathbf{M}_2$, ако двете структури имат един и същи универсум, сигнатурата на $\mathbf{M}_1$ е обогатяване на сигнатурата на $\mathbf{M}_2$ и символите от сигнатурата на $\mathbf{M}_2$ се интерпретират по един и същи начин в двете структури.

Горната дефиниция показва, че ако $\mathbf{M}_1$ е обогатяване на $\mathbf{M}_2$, то всичко, което можем да кажем в $\mathbf{M}_2$, можем да го кажем и в $\mathbf{M}_1$, обаче в структурата $\mathbf{M}_1$ може би можем да казваме и неща, които не могат да се кажат в $\mathbf{M}_2$. Следващото твърдение формализира това наблюдение.

4.2.3. Твърдение. *Нека структурата* $\mathbf{K}$ *е обогатяване на* $\mathbf{M}$ *и* v *е оценка в* $\mathbf{M}$ *(което, разбира се, означава, че* v *е оценка и в* $\mathbf{K}$*). Тогава всеки терм* τ *от сигнатурата на* $\mathbf{M}$ *и всяка формула* φ *от сигнатурата на* $\mathbf{M}$ *са също терм и формула от сигнатурата на* $\mathbf{K}$ *и освен това:*

$$[\![\tau]\!]^{\mathbf{M}} v = [\![\tau]\!]^{\mathbf{K}} v$$
$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{K} \vDash \varphi[v]$$

Доказателство. С тривиална индукция по τ и φ, използвайки това че според дефиницията за обогатяване символите от τ и φ се интерпретират по един и същ начин в двете структури. ∎

✓ **4.2.4.** Да фиксираме сигнатура ВГРАД, която ще съдържа символите, чийто смисъл не се определя от програмиста, а са вградени в езика за програмиране или идват наготово от някоя програмна библиотека. За символите от сигнатурата ВГРАД ще казваме, че са *вградени*. Да

(обджектив си), а структурата — *реализация* или *тяло*.

фиксираме също и структура $\mathbf{B}$, която дава интерпретацията на вградените символи. Когато използваме език за програмиране, ние дефинираме свои собствени функции и предикати, но не променяме смисъла на вградените функции и предикати. Следователно нашата програма дефинира структура, която е обогатяване на $\mathbf{B}$. Когато използваме функционален език за програмиране, това обогатяване ще включва нови функции, а когато използваме логически език за програмиране — нови предикати.

Тъй като искаме да разработим теорията на логическите езици, в които се дефинират само нови предикати, ще обърнем по-специално внимание на този тип обогатявания.

4.2.5. Дефиниция. Когато всички нови символи в дадена обогатена сигнатура или структура са предикатни символи (т.е. не са добавени символи за константи или функционални символи), то тогава такава сигнатура или структура ще наричаме *предикатно обогатяване*.

Псевдоформули

В началото на този раздел казахме, че когато универсумът се състои например от естествените числа, то клауза като $\mathtt{p}(\mathtt{x},\mathtt{x},\mathtt{y})\,\mathtt{:-}$ е еквивалентна на съвкупност от частни случаи от вида $\mathtt{p}(0,0,0)\,\mathtt{:-}$, $\mathtt{p}(0,0,1)\,\mathtt{:-}$, $\mathtt{p}(0,0,2)\,\mathtt{:-}$ и т.н. Обаче тези частни случаи не са клаузи, защото всяка клауза представлява редица от символи, докато използваните тук частни случаи са изрази, включващи елементи на универсума на структурата, в конкретния случай естествени числа. Изрази като $\mathtt{p}(0,0,1)$ дори не са формули. Ще нарачем такива изрази „псевдоформули“.

4.2.6. Дефиниция. *Псевдоформула* при сигнатура $\mathfrak{sig}$ е израз от вида $\mathtt{p}(\beta_1,\beta_2,\ldots,\beta_n)$, където $\mathtt{p}$ е n-местен предикатен символ от сигнатурата $\mathfrak{sig}$, а $\beta_1,\beta_2,\ldots,\beta_n$ са елементи на универсума на $\mathbf{B}$.

Забележка: По принцип би трябвало обектите от горната дефиниция да бъдат наречени „атомарни псевдоформули без функционални символи“. Но тъй като няма да ни трябват по-сложни псевдоформули, ще си позволим за краткост да казваме просто „псевдоформули“.

Верността на псевдоформулите може да се дефинира по очаквания начин:

4.2.7. Дефиниция. Нека структурата $\mathbf{M}$ е обогатяване на $\mathbf{B}$. Ако $\mathtt{p}(\beta_1,\beta_2,\ldots,\beta_n)$ е псевдоформула при сигнатурата на $\mathbf{M}$, то ще казваме,

че тя е *вярна* в структурата $\mathbf{M}$, ако $\mathtt{p}^{\mathbf{M}}(\beta_1, \beta_2, \ldots, \beta_n)$ е истина. Ще записваме това така:

$$\mathbf{M} \vDash \mathtt{p}(\beta_1, \beta_2, \ldots, \beta_n)$$

Време е да дефинираме и какво означава частен случай на атомарна формула. Да обърнем внимание как правим частни случаи на атомарна формула, съдържаща функционални символи. Нека например универсумът отново се състои от естествените числа и функционалният символ + се интерпретира като събиране. Да разгледаме атомарната формула

$$\mathtt{p}(\mathtt{x}, \mathtt{x} + \mathtt{x}) \tag{35}$$

Оказва се, че не е удобно като частни случаи на тази формула да използваме

$$\begin{gathered}\mathtt{p}(0, 0 + 0)\\ \mathtt{p}(1, 1 + 1)\\ \mathtt{p}(2, 2 + 2)\\ \mathtt{p}(3, 3 + 3)\\ \mathtt{p}(4, 4 + 4)\\ \ldots\end{gathered}$$

Като частни случаи трябва да използваме следните псевдоформули::

$$\begin{gathered}\mathtt{p}(0, 0)\\ \mathtt{p}(1, 2)\\ \mathtt{p}(2, 4)\\ \mathtt{p}(3, 6)\\ \mathtt{p}(4, 8)\\ \ldots\end{gathered}$$

4.2.8. Дефиниция. Нека сигнатурата $\mathfrak{sig}$ е предикатно обогатяване на ВГРАД. Ако $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ е атомарна формула при сигнатура $\mathfrak{sig}$ и v е оценка в $\mathbf{B}$ то за псевдоформулата

$$\mathtt{p}([\![\tau_1]\!]^{\mathbf{B}} v, [\![\tau_2]\!]^{\mathbf{B}} v, \ldots, [\![\tau_n]\!]^{\mathbf{B}} v)$$

ще казваме, че е *частен случай* при оценка v на атомарната формула $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$.

Да забележим, че в горната дефиниция се налага обогатяването да бъде предикатно. В противен случай термовете $\tau_1, \tau_2, \ldots, \tau_n$ биха могли да съдържат функционални символи, които не са осмислени в структурата $\mathbf{B}$.

4.2.9. Твърдение. *Нека структурата* $\mathbf{M}$ *е предикатно обогатяване на* $\mathbf{B}$ *и* v *е оценка в* $\mathbf{M}$*. Ако* φ *е атомарна формула при сигнатурата на* $\mathbf{M}$*, то* φ *е вярна в* $\mathbf{M}$ *при оценка* v *тогава и само тогава, когато частният случай на* φ *при оценка* v *е верен в* $\mathbf{M}$*.*

Доказателство. Нека $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$. Съгласно дефиниция 3.3.2 а) φ е вярна в $\mathbf{M}$ при оценка v тогава и само тогава, когато е истина

$$\mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}v, [\![\tau_2]\!]^{\mathbf{M}}v, \ldots, [\![\tau_n]\!]^{\mathbf{M}}v)$$

Тъй като $\mathbf{M}$ е предикатно обогатяване на $\mathbf{B}$, то съгласно твърдение 4.2.3 това се случва тогава и само тогава, когато е истина

$$\mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{B}}v, [\![\tau_2]\!]^{\mathbf{B}}v, \ldots, [\![\tau_n]\!]^{\mathbf{B}}v)$$

Съгласно дефиниция 4.2.7, това е така тогава и само тогава, когато в $\mathbf{M}$ е вярна псевдоформулата

$$\mathtt{p}([\![\tau_1]\!]^{\mathbf{B}}v, [\![\tau_2]\!]^{\mathbf{B}}v, \ldots, [\![\tau_n]\!]^{\mathbf{B}}v)$$

Тази псевдоформула обаче е точно частният случай на φ при оценка v.

■

Да дефинираме и частните случаи на клауза.

4.2.10. Дефиниция. а) *Псевдоклауза* при сигнатура $\mathfrak{sig}$ е израз от вида

$$\varphi \mathrel{:-} \psi_1, \psi_2, \ldots, \psi_n$$

където n може да бъде и 0, а $\varphi, \psi_1, \psi_2, \ldots, \psi_n$ са псевдоформули при сигнатура $\mathfrak{sig}$.

б) Нека сигнатурата $\mathfrak{sig}$ е предикатно обогатяване на ВГРАД и

$$\varphi \mathrel{:-} \psi_1, \psi_2, \ldots, \psi_n$$

е клауза при сигнатура $\mathfrak{sig}$. Ако v е оценка в $\mathbf{B}$, то частен случай на тази клауза при оценка v ще наричаме

$$\varphi' \mathrel{:-} \psi'_1, \psi'_2, \ldots, \psi'_n$$

където $\varphi', \psi'_1, \psi'_2, \ldots, \psi'_n$ са частните случаи при оценка v съответно на $\varphi, \psi_1, \psi_2, \ldots, \psi_n$. Да забележим, че частните случаи на една клауза са псевдоклаузи.

Сега бихме могли да дефинираме и какво означава една псевдоклауза да бъде вярна в структура, но няма да ни се налага да правим това.

Формална дефиниция на правата изводимост с частни случаи

4.2.11. Дефиниция. За едно множество от клаузи Γ ще казваме, че е *логическа програма*, ако всичките му клаузи са от сигнатура, която е предикатно обогатяване на ВГРАД, и отляво на символа :- не се съдържат вградени предикатни символи (т. е. предикатни символи от ВГРАД).

Да си припомним, че при правата изводимост извеждахме новите факти на стъпки. Ако Γ е логическа програма и X са нещата, които сме доказали до момента, то с $T_\Gamma(X)$ ще означим нещата, които можем да докажем на една стъпка от X посредством Γ.

✓ **4.2.12. Дефиниция.** а) Нека Γ е логическа програма. Ще казваме, че псевдоформулата ψ се *извежда едностъпково с частни случаи* от X при програма Γ, ако ψ е вярна в $\mathbf{B}$ или някоя клауза от Γ има частен случай от вида ψ :- $\varphi_1, \varphi_2, \ldots, \varphi_n$, където всяка от псевдоформулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ е елемент на множеството X.

б) Ако Γ е логическа програма, а X — множество от псевдоформули, да означим с $T_\Gamma(X)$ множеството от всички псевдоформули, които се извеждат едностъпково от X при програма Γ.

Една особеност на горната дефиниция, която не бе илюстрирана от неформалния пример за химичен синтез (стр. 215), е това, как извеждаме псевдоформулите с вграден предикатен символ. Да забележим, че в тази дефиниция псевдоформулата ψ може да бъде вярна в $\mathbf{B}$ само ако ψ е с вграден предикатен символ. И също така, някоя клауза от Γ може да има частен случай от вида ψ :- $\varphi_1, \varphi_2, \ldots, \varphi_n$ само ако ψ е с невграден предикатен символ.

4.2.13. Пример. Нека ВГРАД $= \langle +, = \rangle$, универсум на структурата $\mathbf{B}$ са естествените числа, функционалният символ „+“ се интерпретира в $\mathbf{B}$ като събиране, а предикатният символ „=“ като равенство. Нека

$$\Gamma = \{\mathtt{p}(\mathtt{x}+\mathtt{z}, \mathtt{y}+\mathtt{z}) \text{ :- } \mathtt{p}(\mathtt{x}, \mathtt{y})\}$$

и

$$X = \{\mathtt{p}(7, 15)\}$$

Тогава

$$T_\Gamma(X) = \{0=0, 1=1, 2=2, \ldots\} \cup \{\mathtt{p}(7,15), \mathtt{p}(8,16), \mathtt{p}(9,17), \ldots\}$$

Следващото твърдение показва, че операторът T_Γ е монотонен:

✓ **4.2.14. Твърдение.** *Ако $X \subseteq Y$, то $T_\Gamma(X) \subseteq T_\Gamma(Y)$.*

✓ Доказателство. Нека ψ е произволен елемент на $T_\Gamma(X)$. Ако ψ е с вграден предикатен символ, то $\mathbf{B} \vDash \psi$ и значи $\psi \in T_\Gamma(Y)$. Ако пък ψ не е с вграден предикатен символ, то някоя клауза от Γ има частен случай ψ :- $\varphi_1, \varphi_2, \ldots, \varphi_n$, чиито предпоставки $\varphi_1, \varphi_2, \ldots, \varphi_n$ са елементи на X. Но $X \subseteq Y$, следователно всяка от предпоставките е елемент и на Y. Следователно ψ се извежда едностъпково от Y, т. е. $\psi \in T_\Gamma(Y)$. ∎

Нека Γ е множеството от клаузите, с които работим. Да си представим как бихме извеждали нови факти. Ако на стъпка n имаме изведени атомарните формули от множеството X, то на стъпка $n+1$ към тези формули се добавят и формулите от $T_\Gamma(X)$. Следователно формулите, които имаме на стъпка $n+1$, са $X \cup T_\Gamma(X)$. Нека в началото (на стъпка 0) множеството от фактите, които сме извели, е празно:

$$\varnothing$$

След една стъпка множеството от факти, които знаем, става

$$\varnothing \cup T_\Gamma(\varnothing)$$

След още една стъпка към тези факти се добавят още и фактите от $T_\Gamma(\varnothing \cup T_\Gamma(\varnothing))$, следователно новото множество от изведени факти е:

$$\varnothing \cup T_\Gamma(\varnothing) \cup T_\Gamma(\varnothing \cup T_\Gamma(\varnothing))$$

Аналогично, след още една стъпка получаваме

$$\varnothing \cup T_\Gamma(\varnothing) \cup T_\Gamma(\varnothing \cup T_\Gamma(\varnothing)) \cup T_\Gamma(\varnothing \cup T_\Gamma(\varnothing) \cup T_\Gamma(\varnothing \cup T_\Gamma(\varnothing)))$$

И т. н. Ако горните изрази ви изглеждат сложни и все по-сложни, то усещането не е само ваше. Дали не би било по-добре да работим не с $T_\Gamma(X)$, а с $T'_\Gamma(X) = X \cup T_\Gamma(X)$? Ако вместо T_Γ използваме T'_Γ, то горните сметки дават много по-прости изрази. В началото (на стъпка 0) множеството от фактите, които сме извели, е празно:

$$\varnothing$$

След една стъпка множеството от факти, които знаем, става

$$T'_\Gamma(\varnothing)$$

След още една стъпка получаваме

$$T'_\Gamma(T'_\Gamma(\varnothing))$$

Аналогично, след още една стъпка имаме

$$T'_\Gamma(T'_\Gamma(T'_\Gamma(\varnothing)))$$

И т. н. Очевидно при T'_Γ получаваме много по-прости изрази. Това, което може би не е очевидно, е следното: не е нужно да използваме T'_Γ вместо T_Γ, защото горните изрази, използващи T'_Γ, ще си запазят стойността, ако в тях вместо T'_Γ използваме T_Γ. Например

$$T'_\Gamma(T'_\Gamma(T'_\Gamma(\varnothing))) = T_\Gamma(T_\Gamma(T_\Gamma(\varnothing)))$$

Причината за това е следната. От това, че операторът T_Γ е монотонен, следва, че ако $X \subseteq Y$, то $T_\Gamma(X) \subseteq T_\Gamma(Y)$. Ясно е, че

$$\varnothing \subseteq T_\Gamma(\varnothing)$$

Оттук монотонността на T_Γ ни дава

$$T_\Gamma(\varnothing) \subseteq T_\Gamma(T_\Gamma(\varnothing))$$

След това пак заради монотонността получаваме

$$T_\Gamma(T_\Gamma(\varnothing)) \subseteq T_\Gamma(T_\Gamma(T_\Gamma(\varnothing)))$$

Значи имаме следната верига

$$\varnothing \subseteq T_\Gamma(\varnothing) \subseteq T_\Gamma(T_\Gamma(\varnothing)) \subseteq T_\Gamma(T_\Gamma(T_\Gamma(\varnothing))) \subseteq \ldots$$

Да, но от $X \subseteq T_\Gamma(X)$ следва $X \cup T_\Gamma(X) = T_\Gamma(X)$, т. е. $T'_\Gamma(X) = T_\Gamma(X)$. Следователно спокойно можем да работим с T_Γ вместо с T'_Γ.

Видяхме, че $T^n_\Gamma(\varnothing) \subseteq T^{n+1}_\Gamma(\varnothing)$, като $T^n_\Gamma(\varnothing)$ съдържа атомарните формули, които можем да докажем с не повече от n-стъпково доказателство. Значи ако означим

$$T^\infty_\Gamma(\varnothing) = \varnothing \cup T_\Gamma(\varnothing) \cup T_\Gamma(T_\Gamma(\varnothing)) \cup T_\Gamma(T_\Gamma(T_\Gamma(\varnothing))) \cup \ldots$$

то $T^\infty_\Gamma(\varnothing)$ ще бъде множеството от всички неща, които може да се докажат, все едно с колко дълго (но все пак крайно) доказателство.

✓ **4.2.15. Дефиниция.** а) Да положим $T^\infty_\Gamma(\varnothing) = \bigcup_{i=0}^{\infty} T^i_\Gamma(\varnothing)$.

б) Казваме, че φ се *извежда с права изводимост с частни случаи* при програма Γ, ако $\varphi \in T_\Gamma^\infty(\varnothing)$.

Казахме, че $T_\Gamma^\infty(\varnothing)$ съдържа нещата, които могат да се докажат с произволно дълго (но все пак крайно) доказателство. А дали ако допуснем за верни нещата от $T_\Gamma^\infty(\varnothing)$, от тях няма да може да докажем нещо ново? Ако да, то това би означавало, че някои неща не могат да се докажат с крайно доказателство.

За щастие това не се случва. По-точно, ако приложим оператора T_Γ към всички неща, които имат крайно доказателство, то няма да получим нищо ново. Значи

$$T_\Gamma^\infty(\varnothing) = T_\Gamma(T_\Gamma^\infty(\varnothing))$$

Причината, горното равенство да е вярно, е свойство на оператора T_Γ, което се нарича компактност — за да изведем кой да е конкретен елемент на $T_\Gamma(X)$, са ни нужни само краен брой елементи на X. Ако операторът T_Γ не беше компактен*, то редицата

$$\varnothing \subseteq T_\Gamma(\varnothing) \subseteq T_\Gamma(T_\Gamma(\varnothing)) \subseteq T_\Gamma(T_\Gamma(T_\Gamma(\varnothing))) \subseteq \dots$$

би могла да се окаже трансфинитна и бихме могли да я продължим по следния начин:

$$\begin{gathered}\varnothing \subseteq T_\Gamma(\varnothing) \subseteq T_\Gamma(T_\Gamma(\varnothing)) \subseteq T_\Gamma(T_\Gamma(T_\Gamma(\varnothing))) \subseteq \dots \subseteq \\ \subseteq T_\Gamma^\omega(\varnothing) \subseteq T_\Gamma(T_\Gamma^\omega(\varnothing)) \subseteq T_\Gamma(T_\Gamma(T_\Gamma^\omega(\varnothing))) \subseteq T_\Gamma(T_\Gamma(T_\Gamma(T_\Gamma^\omega(\varnothing)))) \subseteq \dots \\ \subseteq T_\Gamma^{\omega.2}(\varnothing) \subseteq T_\Gamma(T_\Gamma^{\omega.2}(\varnothing)) \subseteq T_\Gamma(T_\Gamma(T_\Gamma^{\omega.2}(\varnothing))) \subseteq T_\Gamma(T_\Gamma(T_\Gamma(T_\Gamma^{\omega.2}(\varnothing)))) \subseteq \dots \\ \subseteq \dots \subseteq \dots \subseteq \dots\end{gathered}$$

където

$$\begin{aligned}T_\Gamma^\omega(\varnothing) &= \varnothing \cup T_\Gamma(\varnothing) \cup T_\Gamma(T_\Gamma(\varnothing)) \cup T_\Gamma(T_\Gamma(T_\Gamma(\varnothing))) \cup \dots \\ T_\Gamma^{\omega.2}(\varnothing) &= T_\Gamma^\omega(\varnothing) \cup T_\Gamma(T_\Gamma^\omega(\varnothing)) \cup T_\Gamma(T_\Gamma(T_\Gamma^\omega(\varnothing))) \cup \dots \\ &\dots\end{aligned}$$

✓ **4.2.16. Твърдение.** *а)* $T_\Gamma(T_\Gamma^\infty(\varnothing)) \subseteq T_\Gamma^\infty(\varnothing)$;

б) ако $T_\Gamma(X) \subseteq X$, *то* $T_\Gamma^\infty(\varnothing) \subseteq X$;

в) $T_\Gamma(T_\Gamma^\infty(\varnothing)) = T_\Gamma^\infty(\varnothing)$.

*В математическата логика се разглеждат и изводимости, при които правилата имат безброй много предпоставки. При такива изводимости операторът T_Γ не е компактен.

✓ Доказателство. **(а)** Да допуснем, че $\psi \in T_\Gamma(T_\Gamma^\infty(\varnothing))$. Тогава ψ се извежда едностъпково от $T_\Gamma^\infty(\varnothing)$ при програма Г. Ако ψ е с вграден предикатен символ, то $\mathbf{B} \vDash \psi$, откъдето следва $\psi \in T_\Gamma^\infty(\varnothing)$. Ако пък ψ не е с вграден предикатен символ, то Г съдържа такава клауза ψ :- $\varphi_1, \varphi_2, \ldots, \varphi_n$, че формулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ са от $T_\Gamma^\infty(\varnothing)$. По дефиниция

$$T_\Gamma^\infty(\varnothing) = \bigcup_{i=0}^{\infty} T_\Gamma^i(\varnothing)$$

следователно за всяко $i \in \{1, 2, \ldots, n\}$ съществува такова число k_i, че $\varphi_i \in T_\Gamma^{k_i}(\varnothing)$. Нека $m = \max\{k_1, k_2, \ldots, k_n\}$. Тъй като множествата $T_\Gamma^i(\varnothing)$ се включват едно в друго, то $T_\Gamma^{k_i}(\varnothing) \subseteq T_\Gamma^m(\varnothing)$ за всяко $i \in \{1, 2, \ldots, n\}$ и значи $\varphi_i \in T_\Gamma^m(\varnothing)$, откъдето следва $\psi \in T_\Gamma^{m+1}(\varnothing) \subseteq T_\Gamma^\infty(\varnothing)$.

(б) Ясно е, че

$$\varnothing \subseteq X$$

Оттук монотонността на T_Γ ни дава

$$T_\Gamma(\varnothing) \subseteq T_\Gamma(X) \subseteq X$$

От $T_\Gamma(\varnothing) \subseteq X$ пак от монотонността получаваме

$$T_\Gamma(T_\Gamma(\varnothing)) \subseteq T_\Gamma(X) \subseteq X$$

Продължавайки по същия начин, получаваме $T_\Gamma^i(\varnothing) \subseteq X$ за произволно i. Следователно $T_\Gamma^\infty(\varnothing) \subseteq X$, тъй като $T_\Gamma^\infty(\varnothing)$ е обединение на всички множества от вида $T_\Gamma^i(\varnothing)$.

(в) От $T_\Gamma(T_\Gamma^\infty(\varnothing)) \subseteq T_\Gamma^\infty(\varnothing)$ монотонността ни дава

$$T_\Gamma(T_\Gamma(T_\Gamma^\infty(\varnothing))) \subseteq T_\Gamma(T_\Gamma^\infty(\varnothing))$$

Следователно може да приложим б) при $X = T_\Gamma(T_\Gamma^\infty(\varnothing))$. Получаваме $T_\Gamma^\infty(\varnothing) \subseteq T_\Gamma(T_\Gamma^\infty(\varnothing))$, което заедно с а) ни дава исканото. ■

***Задача 47:** Нека сигнатурата ВГРАД не съдържа нито един предикатен символ и частните случаи на елементите на Г могат да се генерират алгоритмично. Намерете алгоритъм, който генерира елементите на $T_\Gamma^\infty(\varnothing)$.

4.2.17. Задача 47 показва, че в някои случаи е възможно да генерираме алгоритмично елементите на $T_\Gamma^\infty(X)$. Всъщност оказва се, че на

практика много често това наистина е възможно. Следователно, ако φ се извежда с права изводимост с частни случаи от Г, то има начин това да се установи алгоритмично — просто трябва да си генерираме търпеливо елементите на $T_\Gamma^\infty(\varnothing)$, чакайки да се появи φ. Ако φ се извежда с права изводимост с частни случаи от Г, то рано или късно* ще попаднем на φ. Ако обаче φ не се извежда с права изводимост с частни случаи от Г, тогава този алгоритъм ще генерира до безкрайност нови и нови елементи на $T_\Gamma^\infty(\varnothing)$ с надеждата да получи φ, т. е. ще смята безкрайно. За алгоритъм, имащ тези свойства, казваме, че *полурешава* задачата дали някоя атомарна формула се извежда с права изводимост с частни случаи от Г. Този дефект за съжаление е неотстраним — в теория на изчислимостта се доказва, че обикновено не съществува алгоритъм, който *решава* тази задача и изпълнението му винаги завършва.

Понятие за коректност и пълнота

В предния подраздел дефинирахме какво значи една псевдоформула да се извежда при програма Г. Но откъде знаем, че това, което сме дефинирали, работи както трябва? Всъщност отникъде. Дали няма много неща, които следват от клаузите в Г, но не могат да се докажат по начина, използван в предния подраздел? Или може би се случва нещо още по-лошо — с дефинирания метод се извеждат абсурдни неща, които очевидно няма как да са верни? Можем ли да спим спокойно при това положение?

Начинът за възстановяване на спокойствието е следният: първо ще дефинираме какво всъщност искаме. А след това ще проверим математически съответствието между това, което искаме, и това, което имаме.

А какво искаме? Искаме когато някоя псевдоформула се извежда, това да означава, че тя е вярна (в някакъв, все още неясно какъв смисъл). За момента имаме дефиниция какво означава псевдоформула да бъде вярна в структура. Но в коя структура трябва да бъде вярна изведената псевдоформула? Програмата Г не уточнява това. Тя само казва, че някакви клаузи трябва да се верни, но те може да бъдат верни в много структури. Може би тогава трябва да поискаме изведените неща да бъдат верни в структурите, които са модел на програмата Г? Да, това ще ни свърши работа, но със следното уточнение: интересуваме се от онези структури, които не само са модел на Г, но освен това са и обогатявания на $\mathbf{B}$. И така, време е да дадем следната дефиниция:

*По-скоро късно, отколкото рано.

✓ **4.2.18. Дефиниция.** Нека сигнатурата $\mathfrak{sig}$ е предикатно обогатяване на ВГРАД и Г е логическа програма при сигнатура $\mathfrak{sig}$. Ако φ е псевдоформула при сигнатура $\mathfrak{sig}$, която не използва вграден предикатен символ, ще казваме че тя *се удовлетворява* при програма Г, ако φ е вярна във всички структури за $\mathfrak{sig}$, които са едновременно предикатно обогатяване на $\mathbf{B}$ и модел на Г.

Понятието „удовлетворява“ е важно, но очевидно то не е директно проверимо от компютър. Структурите, при които клаузите в Г се оказват верни, могат да бъдат най-разнообразни — толкова разнообразни, че не само компютър, ами и човек не може да ги опише всичките. Как тогава компютър ще може да провери дали във всяка такава структура е вярна псевдоформулата φ? А дори и да се окаже възможно да опишем всички структури, в които клаузите от Г са верни, те най-често ще бъдат безброй много и няма да може с компютър да проверим дали φ е вярна във всяка една от тези безброй много структури. Но да допуснем, че по някакви причини се окаже възможно да се ограничим само с една структура. Ако универсумът ѝ е безкраен, в общия случай задачата пак става нерешима![*]

От друга страна, в предния подраздел дефинирахме и понятието „φ се извежда с права изводимост с частни случаи при програма Г“, което може да се полурешава алгоритмично. Изобщо от никъде не личи каква е връзката между двете понятия „φ се удовлетворява при програма Г“ и „φ се извежда с права изводимост с частни случаи при програма Г“. Дефинициите на тези две понятия са толкова различни една от друга! И въпреки това, оказва се, че те са еквивалентни.

✓ **4.2.19.** Когато някоя изводимост е такава, че всяко нещо, което може да се докаже е вярно, тогава казваме, че тази изводимост е **коректна**. А когато всяко вярно нещо може да се докаже, тогава казваме, че изводимостта е **пълна**.

Например когато кажем, че правата изводимост с частни случаи е коректна, това означава, че ако φ се извежда с права изводимост с частни случаи при програма Г, то φ се удовлетворява при програма Г. А когато кажем, че тя е пълна, това означава, че ако φ се удовлетворява

[*] Например ако универсумът е $\mathbb{N}\backslash\{0\}$, функционалният символ „+“ се интерпретира като събиране, ^ като степенуване и 2 като числото две, то атомарната формула $\mathtt{x}_1\mathtt{\hat{}}(\mathtt{y}_1 + 2) + \mathtt{x}_2\mathtt{\hat{}}(\mathtt{y}_2 + 2) = \mathtt{x}_3\mathtt{\hat{}}(\mathtt{y}_3 + 2)$ е изпълнима тогава и само тогава, когато великата теорема на Ферма е невярна. Няма как да искаме от компютрите да решат задача, която на хората им е отнела векове!

при програма Г, то φ се извежда с права изводимост с частни случаи при програма Г.

Теоремата за коректност и пълнота на дадена изводимост е дълбок резултат, от изключителна важност в математическата логика, който се доказва в почти всеки учебник по логика. Ако погледнем по-философски на тази теорема, тя казва, че едно нещо е доказуемо тогава и само тогава, когато то е вярно. Доказуемост и вярност са две напълно различни понятия и тяхната еквивалентност е нещо удивително!

Да разберем, че нещо е доказуемо, означава да разполагаме с доказателство, което сме в състояние да проверим стъпка по стъпка и което е достатъчно подробно, така че да можем да осъществим проверката изцяло механично, без да се замисляме за каквото и да е. Но и след най-акуратната проверка на едно такова доказателство ние така и няма да разберем защо всъщност доказаното нещо е вярно. От друга страна, да разберем защо нещо е вярно означава да се сдобием с интуиция за това как всъщност стоят нещата, която да ни убеди, че нещото наистина е вярно. Но колкото и добра да е интуицията на един математик, винаги има случаи, когато тя го заблуждава, така че единственият сигурен начин да се убедим във верността на нещо е да имаме негово доказателство. По този начин виждаме, че всяко едно от тези две неща — използването на доказателствата като окончателно и безапелативно свидетелство за вярност и придобиването на интуиция, която „от пръв поглед“ да ни казва как стоят нещата, кое е вярно и кое не — е важно и необходимо в математиката.

Интересно е това, че за тези две неща се грижат напълно различни дялове от главния мозък. Светът на лявото полукълбо е подреден свят, където за всичко си има точни правила. Всяко нещо в този свят кристално ясно и съществува необвързано и само по себе си. За речта — говорима или писмена — отговаря лявото полукълбо. От друга страна, светът на дясното полукълбо е объркан. Всичко в този свят е свързано с всичко и се влияе от всичко. В този свят няма „да“ и „не“, няма „вярно“ и „невярно“, а едно цялостно усещане за съотношението на всяко нещо с всички останали неща. Дясното полукълбо е нямо и вместо реч прави музика.*

Математиката е може би най-точната наука и затова изглежда естествено за нея да отговаря изцяло лявото полукълбо на мозъка. Оказва се обаче, че това съвсем не е така. За формулирането на логически пра-

*Всъщност дясното полукълбо не познава обикновения смисъл на думите, но познава преносния им смисъл и пише стихотворения. Повече подробности за функциите на двете полукълба на мозъка може да се научат напр. от [13].

вилни разсъждения наистина отговаря лявата половина на мозъка, но математическата интуиция и математическите идеи са отдясно. Новата математика се твори от дясната половина на мозъка, а се контролира и проверява от лявата.

Ако четем и заучаваме някое точно математическо доказателство, но интуицията и идеите, скрити в него, ни убягват, това ще бъде безполезно. И също така да четем или слушаме нечии математически идеи, които не са съпроводени с точни доказателства, е все едно да слушаме нечии празни фантазии — на такива фантазии никой математик няма да обърне сериозно внимание (и с право!).

Когато студентите учат някой математически предмет (напр. логическо програмиране), те винаги трябва да търсят скритите идеи и интуицията, обясняващи смисъла дефинициите и изясняващи доказателствата. Също така те трябва да изучават как могат да изказват тези идеи на точен математически език. И когато това стане, те ще могат да се движат свободно в абстрактния свят на математиката и да творят, знаейки, че винаги ще бъдат в състояние да облекат измислените неща в точен математически формализъм, който да им гарантира, че не са измислили някоя глупост и с който ще могат да споделят идеите си. Ако студентите учат нещата наизуст, без всъщност да ги разбират, те най-вероятно ще си вземат изпита, но неприятните усещания, които са изпитвали докато учат, могат да предизвикат у един съпричастен преподавател единствено съчувствие и съжаление. Студенти пък, които са се опитвали да учат идеите без да проследяват доказателствата, са направили нещо безполезно, което е може би по-подходящо за философски факултет, отколкото за ФМИ. Обаче студенти, които са успели да схванат истински нещата, са радост за преподавателите, защото такива студенти са успели поне донякъде да видят красотите в чудния, хем измислен, хем истински свят, в който преподавателите са се опитали да ги заведат.

Забележка: Можем накратко да резюмираме казаното до тук само с едно изречение: „Използвайте целия си мозък, а не само едната му половина!“

Положителна диаграма на структура

В дефиницията на понятието „φ се удовлетворява при програма Γ“ удовлетворяването на φ се установява посредством верността на φ в определени структури. В дефиницията на понятието „φ се извежда с права изводимост с частни случаи при програма Γ“ пък, извеждането

на φ се установява посредством принадлежността на φ на определено множество от псевдоформули. Следователно, ако искаме да свържем тези две понятия, имаме нужда от връзка между структурите и множествата от псевдоформули. Положителната диаграма на една структура ни дава тази връзка.

4.2.20. Дефиниция. Нека структурата $\mathbf{M}$ е предикатно обогатяване на $\mathbf{B}$. Множеството $\mathrm{diag}(\mathbf{M})$ от всички верни в $\mathbf{M}$ псевдоформули се нарича *положителна диаграма* на $\mathbf{M}$.

И така, тази дефиниция ни дава еднопосочна връзка между структури и множества — за всяка структура $\mathbf{M}$ имаме множество $\mathrm{diag}(\mathbf{M})$. Ще ни трябва и обратната посока — по дадено множество да намерим структура, чиято положителна диаграма е това множество. Следващото твърдение ни дава тази обратна посока. Ако $\mathbf{M}$ е предикатно обогатяване на $\mathbf{B}$, то тогава структурата $\mathbf{B}$ определя еднозначно кой е универсумът на $\mathbf{M}$ и как в $\mathbf{M}$ се интерпретират символите за индивидни константи и функционалните символи. Трябва да се определи само интерпретацията на предикатните символи.

4.2.21. Твърдение. *Нека сигнатурата* $\mathfrak{sig}$ *е предикатно обогатяване на* ВГРАД *и* D *е такова множество от псевдоформули за сигнатурата* $\mathfrak{sig}$*, че за всяка псевдоформула* φ *от сигнатурата* ВГРАД

$$\varphi \in D \longleftrightarrow \mathbf{B} \vDash \varphi$$

Тогава съществува единствена структура за сигнатурата $\mathfrak{sig}$*, която е предикатно обогатяване на* $\mathbf{B}$ *и чиято положителна диаграма е* D*.*

Доказателство. **(съществуване)** Нека $\mathbf{M}$ е с универсум — универсумът на $\mathbf{B}$, и символите за индивидни константи и функционалните символи се интерпретират в $\mathbf{M}$ по същия начин, както и в $\mathbf{B}$. Ако пък $\mathtt{p}$ е n-местен предикатен символ, то нека по дефиниция

$$\mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)$$

е истина тогава и само тогава, когато

$$\mathtt{p}(\mu_1, \mu_2, \ldots, \mu_n) \in D$$

Тогава е ясно, че $\mathrm{diag}(\mathbf{M}) = D$. Освен това

$$\mathbf{M} \vDash \mathtt{p}(\mu_1, \mu_2, \ldots, \mu_n) \longleftrightarrow \mathtt{p}(\mu_1, \mu_2, \ldots, \mu_n) \in D \longleftrightarrow \mathbf{B} \vDash \mathtt{p}(\mu_1, \mu_2, \ldots, \mu_n)$$

следователно $\mathbf{M}$ е предикатно обогатяване на $\mathbf{B}$.

(единственост) Нека $\mathbf{K}$ е предикатно обогатяване на $\mathbf{B}$ и $\mathrm{diag}(\mathbf{K}) = D$. Щом $\mathbf{K}$ е обогатяване на $\mathbf{B}$, то значи $\mathbf{K}$ има същия универсум както и $\mathbf{B}$, а значи и както $\mathbf{M}$. По същата причина символите от ВГРАД се интерпретират в $\mathbf{K}$, както в $\mathbf{B}$, а значи както и в $\mathbf{M}$. Ако пък $\mathtt{p}$ е невграден предикатен символ, то

$$\mathtt{p}^{\mathbf{K}}(\mu_1, \mu_2, \ldots, \mu_n)$$

е истина тогава и само тогава, когато

$$\mathtt{p}(\mu_1, \mu_2, \ldots, \mu_n) \in \mathrm{diag}(\mathbf{K}) = \mathrm{diag}(\mathbf{M}) = D$$

т. е. тогава и само тогава, когато

$$\mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)$$

е истина. ∎

Структурите, с които работим, са модели на някакво множество от клаузи Γ (нашата програма). Следващата лема превежда това свойство на езика на положителните диаграми.

4.2.22. Лема. *Нека структурата* $\mathbf{M}$ *е предикатно обогатяване на* $\mathbf{B}$*, а* Γ *е логическа програма при сигнатурата на* $\mathbf{M}$*. Тогава структурата* $\mathbf{M}$ *е модел на* Γ *тогава и само тогава, когато* $T_\Gamma(\mathrm{diag}(\mathbf{M})) \subseteq \mathrm{diag}(\mathbf{M})$*.*

Доказателство. Да припомним, че съгласно твърдение 4.2.9 една атомарна формула е вярна в $\mathbf{M}$ при оценка v тогава и само тогава, когато частният ѝ случай при оценка v е верен в $\mathbf{M}$. От своя страна, частният случай е верен в $\mathbf{M}$ тогава и само тогава, когато е елемент на $\mathrm{diag}(\mathbf{M})$.

Структурата $\mathbf{M}$ е модел на Γ тогава и само тогава, когато клаузите от Γ са верни в $\mathbf{M}$ при всяка оценка. А това е така тогава и само тогава, когато за всяка клауза ψ :- $\varphi_1, \varphi_2, \ldots, \varphi_n$ от Γ и всяка оценка v е вярно съждението

> *Ако* $\varphi_1, \varphi_2, \ldots, \varphi_n$ *са верни в* $\mathbf{M}$ *при оценка* v*, то* ψ *е вярна в* $\mathbf{M}$ *при оценка* v*.*

Съгласно наблюдението, което направихме в началото на доказателството, това съждение е вярно за произволна клауза от Γ и произволна оценка v тогава и само тогава, когато за всеки частен случай ψ' :- $\varphi'_1, \varphi'_2, \ldots, \varphi'_n$ на клауза от Γ е вярно съждението

Ако всяка от псевдоформулите $\varphi'_1, \varphi'_2, \ldots, \varphi'_n$ *е елемент на* $\mathrm{diag}(\mathbf{M})$*, то* ψ' *също е елемент на* $\mathrm{diag}(\mathbf{M})$.

Тъй като предикатният символ на ψ' не е от ВГРАД, то горното съждение е вярно за всички частни случаи на елементи на Γ тогава и само тогава, когато псевдоформулите с невграден предикатен символ, които се извеждат едностъпково от $\mathrm{diag}(\mathbf{M})$ при програма Γ, са елементи на $\mathrm{diag}(\mathbf{M})$. Последното е така тогава и само тогава, когато $T_\Gamma(\mathrm{diag}(\mathbf{M})) \subseteq \mathrm{diag}(\mathbf{M})$, защото от това, че $\mathbf{M}$ е предикатно обогатяване на $\mathbf{B}$, следва, че за всяка псевдоформула ψ с вграден предикатен символ, $\psi \in T_\Gamma(\mathrm{diag}(\mathbf{M})) \longleftrightarrow \mathbf{B} \vDash \psi \longleftrightarrow \mathbf{M} \vDash \psi \longleftrightarrow \psi \in \mathrm{diag}(\mathbf{M})$. ∎

Най-малък модел на логическа програма

Когато пишем логическа програма, ние си мислим, че в нея дефинираме някакви предикати. С други думи, нашата логическа програма дефинира определена структура. Какви свойства има тази структура? От една страна искаме в тази структура вградените предикатни символи да се интерпретират както в $\mathbf{B}$. Това означава, че структурата, която дефинираме, трябва да бъде предикатно обогатяване на $\mathbf{B}$. От друга страна, в тази структура трябва да са тъждествено верни клаузите от програмата. Достатъчни ли са тези две изисквания, за да определят еднозначно структурата, която имаме предвид? Следващият пример показва, че не.

4.2.23. Пример. Нека структурата $\mathbf{M}$ има същия универсум като $\mathbf{B}$ и символите за константи, функционалните символи и вградените предикатни символи се интерпретират също като в структурата $\mathbf{B}$. Нека невградените предикатни символи са винаги истина в $\mathbf{M}$ (при произволни аргументи). По дефиниция структурата $\mathbf{M}$ ще бъде предикатно обогатяване на $\mathbf{B}$. Освен това може да се забележи, че тя ще бъде модел не само за нашата логическа програма, ами изобщо за всички логически програми. Наистина, нека

$$\psi \text{ :- } \varphi_1, \varphi_2, \ldots, \varphi_n$$

е произволна клауза, такава че предикатният символ на атомарната формула ψ не е вграден. Тогава формулата ψ ще бъде вярна в $\mathbf{M}$ при произволна оценка, а от тук следва, че и цялата клауза ще бъде вярна вярна при произволна оценка.

Но въпреки, че една логическа програма може да има много модели, които са обогатявания на $\mathbf{B}$, все пак когато пишем програмата,

ние имаме предвид един точно определен модел. Нека например сигнатурата ВГРАД съдържа триместен предикатен символ `append` и нека структурата $\mathbf{B}$ интерпретира този предикатен символ също като пролог, т.е. $\texttt{append}(x, y, z)$ означава „списъкът z е конкатенация на списъците x и y“. Да разгледаме логическата програма, която се състои от следната клауза:*

$$\texttt{p(x, y) :- append(x, x, y)}$$

Интуитивно ни се иска да считаме, че тази програма дефинира предикат $p(x, y)$, който казва, че ако конкатенираме списъка x със себе си, ще получим списъка y.

Как можем да дефинираме формално структурата, „която имаме предвид“? Едно очевидно свойство на „лошата“ структура $\mathbf{M}$ от пример 4.2.23 е това, че в нея има твърде много верни неща. Всъщност измежду всички структури, които са едновременно обогатявания на $\mathbf{B}$ и модели на логическата програма, в структурата $\mathbf{M}$ ще бъдат верни възможно най-много псевдоформули. Когато обаче един програмист пише логическа програма, той иска за дефинираните от него предикати да е истина само това, което изрично е посочено в програмата като истина, а всичко останало да е лъжа. Възниква въпросът: не може ли да дефинираме структура, която също е обогатяване на $\mathbf{B}$ и която също е модел на логическата програма, но в която са верни възможно най-малко псевдоформули? Оказва се, че отговорът на този въпрос е положителен. Тази структура се нарича *най-малък модел* на логическата програма. Именно най-малкият модел е структурата, която програмистът има предвид, когато пише логическа програма.

4.2.24. Теорема за най-малкия модел. *Нека* Γ *е логическа програма при сигнатура* $\mathfrak{sig}$. *Тогава съществува такава структура* $\mathbf{M}$ *за* $\mathfrak{sig}$, *че*

а) $\mathrm{diag}(\mathbf{M}) = T_\Gamma^\infty(\varnothing)$;

б) $\mathbf{M}$ *е предикатно обогатяване на* $\mathbf{B}$ *и модел на* Γ;

в) ако структурата $\mathbf{K}$ *е предикатно обогатяване на* $\mathbf{B}$ *и модел на* Γ, *то* $\mathrm{diag}(\mathbf{M}) \subseteq \mathrm{diag}(\mathbf{K})$.

Доказателство. **(а)** Съгласно твърдение 4.2.21, съществува структура $\mathbf{M}$ за $\mathfrak{sig}$ (при това единствена), която е предикатно обогатяване на $\mathbf{B}$ и $\mathrm{diag}(\mathbf{M}) = T_\Gamma^\infty(\varnothing)$.

*Тук не се съобразяваме с изискването на пролог, според което променливите трябва да започват с главна буква.

(б) Вече установихме, че $\mathbf{M}$ е предикатно обогатяване на $\mathbf{B}$. Съгласно твърдение 4.2.16 а), $T_\Gamma(T_\Gamma^\infty(\varnothing)) \subseteq T_\Gamma^\infty(\varnothing)$. Според лема 4.2.22 това ни дава, че $\mathbf{M}$ е модел на Γ.

(в) Нека $\mathbf{K}$ е предикатно обогатяване на $\mathbf{B}$ и модел на Γ. Съгласно лема 4.2.22, $T_\Gamma(\mathrm{diag}(\mathbf{K})) \subseteq \mathrm{diag}(\mathbf{K})$. Според твърдение 4.2.16 б) това означава, че $\mathrm{diag}(\mathbf{M}) = T_\Gamma^\infty(\varnothing) \subseteq \mathrm{diag}(\mathbf{K})$. ∎

Теореми за коректност и пълнота на правата изводимост с частни случаи

Почти всички доказателства за коректност в математическата логика се доказват по следния начин — първо показваме, че след едностъпково доказателство от верни неща получаваме пак верни неща. Щом след една стъпка, ще получим верни неща, значи след още една пак ще получим верни неща и т. н. С други думи, всички неща, които могат да се изведат, са верни и значи имаме коректност.

В следващото доказателство твърдението, че след едностъпково доказателство от верни неща получаваме пак верни неща, изглежда така: $T_\Gamma(\mathrm{diag}(\mathbf{M})) \subseteq \mathrm{diag}(\mathbf{M})$. Твърдението пък, че всички неща, които могат да се изведат, са верни, изглежда така: $T_\Gamma^\infty(\varnothing) \subseteq \mathrm{diag}(\mathbf{M})$.

4.2.25. Теорема за коректност на правата изводимост с частни случаи. *Нека Γ е логическа програма. Ако някоя псевдоформула се извежда с права изводимост с частни случаи при програма Γ, то тя се удовлетворява при програма Γ.*

Доказателство. Нека φ се извежда с права изводимост с частни случаи при програма Γ. По дефиниция това означава, че $\varphi \in T_\Gamma^\infty(\varnothing)$ и значи φ не използва вграден предикатен символ.

Трябва да докажем, че ако $\mathbf{M}$ е структура за сигнатурата на Γ, която е едновременно обогатяване на $\mathbf{B}$ и модел на Γ, то $\mathbf{M}$ е модел на φ. Но щом $\mathbf{M}$ е модел на Γ, то от лема 4.2.22 получаваме, че $T_\Gamma(\mathrm{diag}(\mathbf{M})) \subseteq \mathrm{diag}(\mathbf{M})$, и значи твърдение 4.2.16 б) ни дава $T_\Gamma^\infty(\varnothing) \subseteq \mathrm{diag}(\mathbf{M})$. Но $\varphi \in T_\Gamma^\infty(\varnothing)$, следователно $\varphi \in \mathrm{diag}(\mathbf{M})$, и значи $\mathbf{M}$ е модел на φ. ∎

За да докажем теоремата за пълнота на правата изводимост с частни случаи, ще се възползваме от най-малкия модел. Доказателството ще извършим по следния начин. Грубо казано, искаме да видим, че ако псевдоформулата φ е вярна в моделите на Γ, то φ може да се изведе с права изводимост. Но щом φ е вярна в моделите на Γ, то в частност формулата φ ще бъде вярна и в най-малкия модел на Γ. Най-малкият

модел има това интересно свойство, че него са верни точно нещата, които могат да се изведат от Γ. Значи щом φ е вярна в най-малкия модел, то φ може да се изведе.

4.2.26. Теорема за пълнота на правата изводимост с частни случаи. *Нека* Γ *е логическа програма. Ако някоя псевдоформула се удовлетворява при програма* Γ*, то тя се извежда с права изводимост с частни случаи при програма* Γ*.*

Доказателство. Нека $\mathbf{M}$ е структурата от теоремата най-малкият модел. Тя е модел на Γ и предикатно обогатяване на $\mathbf{B}$. Ако φ е псевдоформула, която се удовлетворява при програма Γ, то φ не използва вграден предикатен символ и $\mathbf{M}$ е модел на φ. Следователно $\varphi \in \mathrm{diag}(\mathbf{M})$. Но съгласно теоремата за най-малкия модел $\mathrm{diag}(\mathbf{M}) = T_\Gamma^\infty(\varnothing)$ и значи $\varphi \in T_\Gamma^\infty(\varnothing)$, което пък означава, че φ се извежда с права изводимост с частни случаи при програма Γ. ∎

4.3. Обратна изводимост

Обратна изводимост с частни случаи

За разлика от правата изводимост, при обратната изводимост използването на частни случаи (засега) няма практически приложения. Въпреки това сега ще разработим теорията на обратната изводимост с частни случаи. Ще направим това, защото е неразумно без подготовка да се гмуркаме изведнъж в дълбокото, тъй като в противен случай може и да не изплуваме. Дори да се интересуваме не от абстрактна теория, от приложенията ѝ, пак има смисъл преди да се впускаме да разработваме теорията на смислената практика, да разработим теория, която макар и да е с безсмислена практика, обаче е по-проста и по-лесна за разбиране.

При правата изводимост използваме клаузите от Γ, които „знаем“, за да доказваме все повече и повече неща — елементите на $T_\Gamma^\infty(\varnothing)$ — и правим това докато успеем да получим желаната цел. При обратната изводимост започваме не с нещата, които знаем, а със запитването, което искаме да докажем, и постепенно го свеждаме до по-лесни за доказване запитвания. Запитванията, които се опитваме да докажем, не са атомарни формули, а конюнкции от атомарни формули, които в дефиниция 4.1.3 нарекохме „запитвания“. При обратната изводимост с частни случаи се използват частни случаи на запитвания, т.е. псевдозапитвания. Да дефинираме какво значи псевдозапитване.

4.3.1. Дефиниция. а) *Псевдозапитване* при сигнатура $\mathfrak{sig}$ е израз от вида

$$\texttt{?-}\ \varphi_1, \varphi_2, \ldots, \varphi_n$$

където $\varphi_1, \varphi_2, \ldots, \varphi_n$ са псевдоформули при сигнатура $\mathfrak{sig}$. Когато $n = 0$ псевдозапитването се нарича *празно псевдозапитване*.

б) Нека структурата $\mathbf{M}$ е обогатяване на $\mathbf{B}$. Едно псевдозапитване е *вярно* в $\mathbf{M}$, ако всичките му псевдоформули са верни в $\mathbf{M}$.

в) Нека Г е логическа програма. Едно псевдозапитване се *удовлетворява* при програма Г, ако всичките му псевдоформули се удовлетворяват при програма Г.

И тъй, при обратната изводимост започваме с някакво псевдозапитване $\texttt{?-}\ \varphi_1, \varphi_2, \ldots, \varphi_n$ и го преработваме, използвайки клаузите от програмата. Да дефинираме как се прави това.

4.3.2. Дефиниция. Нека Г е логическа програма.

✓ а) Казваме че псевдозапитването

$$\texttt{?-}\ \varphi_0, \varphi_1, \varphi_2 \ldots, \varphi_n$$

се свежда едностъпково с обратна изводимост с частни случаи по първи начин към

$$\texttt{?-}\ \psi_1, \psi_2, \ldots, \psi_k, \varphi_1, \varphi_2 \ldots, \varphi_n$$

при програма Г, ако псевдоклаузата $\varphi_0\ \texttt{:-}\ \psi_1, \psi_2, \ldots, \psi_k$ е частен случай на клауза от Г (разбира се тук позволяваме $k = 0$).

б) Казваме че псевдозапитването

$$\texttt{?-}\ \varphi_0, \varphi_1, \varphi_2 \ldots, \varphi_n$$

се свежда едностъпково с обратна изводимост с частни случаи по втори начин към

$$\texttt{?-}\ \varphi_1, \varphi_2 \ldots, \varphi_n$$

при програма Г, ако φ_0 е псевдоформула при сигнатурата ВГРАД, която е вярна в структурата $\mathbf{B}$.

✓ **4.3.3. Дефиниция.** а) Казваме, че псевдозапитването δ *се свежда с обратна изводимост с частни случаи* при програма Г към δ', ако съществува такава редица от псевдозапитвания $\delta_0, \delta_1, \delta_2, \ldots, \delta_m$, че първото псевдозапитване δ_0 е δ, всяко псевдозапитване се свежда едностъпково към следващото и последното псевдозапитване δ_m е δ'.

б) Казваме, че псевдозапитването δ се извежда с обратна изводимост с частни случаи при програма Г, ако δ се свежда към празното псевдозапитване при програма Г.

Да забележим, че всяко псевдозапитване се свежда към себе си. Релацията „свеждане“ представлява рефлексивно и транзитивно затваряне на релацията „едностъпково свеждане“.

4.3.4. Лема. *Нека структурата* $\mathbf{M}$ *е модел на логическата програма* Г *и псевдозапитването* δ *се свежда едностъпково с обратна изводимост при програма* Г *към* δ'. *Ако* $\mathbf{M} \vDash \delta'$, *то* $\mathbf{M} \vDash \delta$.

Доказателство. Да допуснем, че $\mathbf{M} \vDash \delta'$. Трябва да докажем, че $\mathbf{M} \vDash \delta$. Нека псевдозапитването δ е

$$\texttt{?-}\ \varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n$$

Ще разгледаме два случая в зависимост от това по кой начин δ се свежда към δ'.

Ако това става по първи начин (вж. дефиниция 4.3.2), тогава псевдозапитването δ' има вида

$$\texttt{?-}\ \psi_1, \psi_2, \ldots, \psi_k, \varphi_1, \varphi_2, \ldots, \varphi_n$$

и Г съдържа клауза

$$\varphi_0' \texttt{:-}\ \psi_1', \psi_2', \ldots, \psi_k'$$

с частен случай

$$\varphi_0 \texttt{:-}\ \psi_1, \psi_2, \ldots, \psi_k$$

Нека този частен случай се получава при оценка v. Щом $\mathbf{M} \vDash \delta'$, то в $\mathbf{M}$ са верни псевдоформулите $\psi_1, \psi_2, \ldots, \psi_k, \varphi_1, \varphi_2, \ldots, \varphi_n$. Щом в $\mathbf{M}$ са верни псевдоформулите $\psi_1, \psi_2, \ldots, \psi_k$, то значи формулите $\psi_1', \psi_2', \ldots, \psi_k'$ са верни в $\mathbf{M}$ при оценка v. Щом $\mathbf{M}$ е модел на Г, то в частност $\mathbf{M}$ е модел и на клаузата $\varphi_0' \texttt{:-}\ \psi_1', \psi_2', \ldots, \psi_k'$, откъдето следва, че в $\mathbf{M}$ формулата φ_0' е вярна при оценка v, от където пък следва, че в $\mathbf{M}$ е вярна псевдоформулата φ_0. Така получаваме, че $\mathbf{M}$ е модел на всяка от псевдоформулите $\varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n$, а значи и на псевдозапитването δ.

Ако δ се свежда едностъпково към δ' по втори начин, тогава δ' има вида

$$\texttt{?-}\ \varphi_1, \varphi_2, \ldots, \varphi_n$$

като псевдоформулата φ_0 е с вграден предикатен символ и е вярна в $\mathbf{B}$. Щом φ_0 е вярна в $\mathbf{B}$ и $\mathbf{M}$ е обогатяване на $\mathbf{B}$, то φ_0 е вярна и в $\mathbf{M}$.

Псевдоформулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ пък са верни в $\mathbf{M}$, защото са част от псевдозапитването δ', което е вярно в $\mathbf{M}$. Излиза, че $\mathbf{M}$ е модел на всяка от псевдоформулите $\varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n$, а значи и на псевдозапитването δ. ■

4.3.5. Теорема за коректност на обратната изводимост с частни случаи. *Ако δ се извежда с обратна изводимост с частни случаи при програма Γ, то δ се удовлетворява при програма Γ.*

Доказателство. Нека структурата $\mathbf{M}$ е модел на Γ. Нека редицата, която показва, че δ се свежда с обратна изводимост с частни случаи към празното запитване при програма Γ е $\delta_0, \delta_1, \delta_2, \ldots, \delta_m$. Тъй като в тази редица последното запитване е празното запитване, то $\mathbf{M}$ е негов модел. Тъй като всяко запитване се свежда едностъпково към следващото, от предната лема следва, че $\mathbf{M}$ е модел и на предпоследното запитване, от където получаваме, че $\mathbf{M}$ е модел и на пред-предпоследното и т. н. Значи $\mathbf{M}$ е модел и на първото запитване, т. е. на δ. ■

Пристъпваме към доказателството на теоремата за пълнота на обратната изводимост с частни случаи. Идеята на доказателството принадлежи на проф. Димитър Скордев.

4.3.6. Дефиниция. Нека Γ е множество от клаузи. Казваме, че псевдоформулата φ е *отстранима* при програма Γ, ако всяко псевдозапитване от вида

$$\texttt{?-}\ \varphi, \psi_1, \psi_2, \ldots, \psi_n.$$

(позволяваме $n = 0$) се свежда с обратна изводимост с частни случаи при програма Γ към псевдозапитването

$$\texttt{?-}\ \psi_1, \psi_2, \ldots, \psi_n.$$

4.3.7. Лема. *Нека Γ е множество от клаузи и $\varphi \in T_\Gamma^\infty(\varnothing)$. Тогава φ е отстранима при програма Γ.*

Доказателство. С индукция по n ще докажем, че лемата е вярна когато $\varphi \in T_\Gamma^n(\varnothing)$.

Когато $n = 0$, няма какво да доказваме, защото $T_\Gamma^0(\varnothing) = \varnothing$.

Да допуснем, че лемата е вярна за n. Нека $\varphi \in T_\Gamma^{n+1}(\varnothing)$, т.е. φ се извежда с частни случаи от $T_\Gamma^n(\varnothing)$. Единият начин това да се случи (вж. дефиниция 4.2.12), е когато φ е псевдоформула при сигнатурата ВГРАД, която е вярна в $\mathbf{B}$. Но тогава отстранимостта на φ следва

непосредствено от дефиницията за едностъпково свеждане (дефиниция 4.3.2). Другият начин това да се случи е някоя клауза от Γ да има такъв частен случай

$$\varphi \texttt{ :- } \chi_1, \chi_2, \chi_3, \chi_4, \ldots, \chi_n \tag{36}$$

че $\chi_1, \chi_2, \ldots, \chi_n \in T_\Gamma^n(\varnothing)$. От индукционното предположение имаме, че псевдоформулите $\chi_1, \chi_2, \ldots, \chi_n$ са отстраними. За да докажем, че φ е отстранима, да изберем произволно псевдозапитване от вида

$$\texttt{?- } \varphi, \psi_1, \psi_2, \ldots, \psi_m$$

Използвайки псевдоклаузата (36), виждаме, че това псевдозапитване се свежда едностъпково с обратна изводимост с частни случаи към псевдозапитването

$$\texttt{?- } \chi_1, \chi_2, \chi_3, \chi_4, \ldots, \chi_n, \psi_1, \psi_2, \ldots, \psi_m$$

Вече видяхме, че псевдоформулата χ_1 е отстранима, значи горното псевдозапитване се свежда (незадължително едностъпково) към

$$\texttt{?- } \chi_2, \chi_3, \chi_4 \ldots, \chi_n, \psi_1, \psi_2, \ldots, \psi_m$$

Псевдоформулата χ_2 също е отстранима и значи това пък псевдозапитване можем да сведем до

$$\texttt{?- } \chi_3, \chi_4, \ldots, \chi_n, \psi_1, \psi_2, \ldots, \psi_m$$

По този начин, една по една можем да отстраним всички псевдоформули χ_i и да получим

$$\texttt{?- } \psi_1, \psi_2, \ldots, \psi_m.$$

■

4.3.8. Теорема за пълнота на обратната изводимост с частни случаи. *Ако едно псевдозапитване се удовлетворява при програма* Γ*, то то се извежда с обратна изводимост с частни случаи при програма* Γ*.*

Доказателство. Да допуснем, че псевдозапитването $\texttt{?- } \varphi_1, \ldots, \varphi_n$ се удовлетворява при програма Γ. От теоремата за пълнота на правата изводимост с частни случаи (теорема 4.2.26) следва, че псевдоформулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ са елементи на $T_\Gamma^\infty(\varnothing)$, и значи тези псевдоформули са отстраними. Щом φ_1 е отстранима, то значи горното псевдозапитване може да се сведе към $\texttt{?- } \varphi_2, \varphi_3, \ldots, \varphi_n$. Щом φ_2 е отстранима, то

последното псевдозапитване пък може да се сведе към ?- $\varphi_3, \varphi_4, \ldots, \varphi_n$. Продължавайки по този начин, виждаме, че първоначалното псевдозапитване може да се сведе към празното псевдозапитване, което по дефиниция означава, че то се извежда с обратна изводимост с частни случаи при програма Г. ■

Логическо програмиране с ограничения

Да видим сега как можем да реализираме обратната изводимост без да се използват частни случаи. Идеята е следната: ще използваме запитвания с ограничения, всяко от които е еквивалентно на безкрайна съвкупност от псевдозапитвания. Например безкрайната съвкупност от псевдозапитванията

$$\begin{gathered}\texttt{?- p("0", "1")}\\ \texttt{?- p("1", "2")}\\ \texttt{?- p("2", "3")}\\ \texttt{?- p("3", "4")}\\ \ldots\end{gathered}$$

е еквивалентна на

$$\langle \texttt{?- p(x, y)} \parallel \texttt{y} = \texttt{x} + 1 \rangle$$

Изразът след двойната вертикална черта се нарича „ограничение“.* Ще предполагаме, че имаме алгоритъм за решаване на ограниченията.

Да поясним нещата с точна дефиниция.

✓ **4.3.9. Дефиниция.** а) *Ограничение* ще наричаме формула, която или е равна на $\top$, или представлява конюнкция от атомарни формули от сигнатурата ВГРАД. Ако условно приемем, че $\top$ е конюнкция на нула атомарни формули от сигнатурата ВГРАД, то тогава може да кажем, че ограничение означава конюнкция на нула или повече атомарни формули от ВГРАД. За ограничението $\varphi_1 \,\&\, \varphi_2 \,\&\, \cdots \,\&\, \varphi_n$ ще използваме следния запис:

$$\varphi_1, \varphi_2, \ldots, \varphi_n$$

б) *Състоянието* е формула от вида $\delta \,\&\, \varepsilon$, където δ е запитване, а ε — ограничение. За състоянието $\delta \,\&\, \varepsilon$ ще използваме следния запис:

$$\langle \delta \parallel \varepsilon \rangle$$

*На английски „constraint“.

✓ **4.3.10.** Езиците за логическо програмиране с ограничения работят по следния начин. Да допуснем, че сме задали на компютъра запитване δ. Започваме изпълнението на логическата програма от състояние $\langle \delta \parallel \top \rangle$. След това на всяка стъпка преобразуваме текущото състояние по начина, описан в следващата дефиниция, като идеята на това преобразуване е следната — постепенно превеждаме нещата, които са поискани в запитването δ, като използваме само предикати от $\mathbf{B}$. По този начин все по-голяма и по-голяма част от информацията, съдържаща се в състоянието, ще се премества от запитването в ограничението. Ако след краен брой такива преобразувания получим състояние от вида $\langle \texttt{?-} \parallel \varepsilon \rangle$, т.е. състояние, в което запитването съдържа нула формули, то значи цялата информация вече се съдържа в ограничението. Значи единственото, от което имаме нужда, е алгоритъм, с помощта на който можем да проверим дали така полученото ограничение ε е изпълнимо.

4.3.11. И така, да фиксираме някакъв алгоритъм, който може да бъде прилаган към произволно ограничение (при сигнатурата, с която работим), и който без да се зацикля връща един от следните три отговора:

- ограничението е неизпълнимо;
- ограничението е изпълнимо;
- не може да се установи дали ограничението е изпълнимо.

За един такъв алгоритъм казваме, че е пълен, ако той винаги успява да отговори дали ограничението е изпълнимо.

Разбира се при различните структури $\mathbf{B}$ има различни вградени предикати и значи трябва да се използват различни алгоритми за решаване на ограниченията. Следователно в момента няма как да уточним какъв точно е алгоритъмът за решаване на ограниченията. В раздел 4.5 ще разгледаме най-важния алгоритъм за решаване на ограничения, който се използва в езика пролог.

Почти винаги използваните на практика алгоритми са в състояние не само да установят дали дадено ограничение е изпълнимо, но също така когато установят, че ограничението е изпълнимо, те намират и конкретни стойности на променливите, при които ограничението е вярно.

4.3.12. От тук нататък ще предполагаме, че сигнатурата ВГРАД съдържа предикатен символ $=$, който в $\mathbf{B}$ се интерпретира като равенство. Ще направим това, за да си опростим малко нещата, а и защото в популярния език за програмиране пролог има такъв вграден предикатен символ. Все пак добре е да имаме предвид следните две неща:

– Има начин нещата да се направят и без да се използва равенството.*
– Обикновено равенството на два безкрайни обекта не може да се установява алгоритмично. Следователно ако искаме в логическата програма да използваме безкрайни типове данни (каквито има напр. в ленивите функционални езици), тогава трябва да работим без равенство.

Преди да дефинираме как работи обратната изводимост с ограничения, ще имаме нужда от следната помощна дефиниция:

4.3.13. Дефиниция. а) Една субституция е *преименуваща*, ако заменя променливи с променливи по биективен начин.

б) Нека φ е клауза. Всички клаузи от вида φs, където s е преименуваща субституция, се наричат *варианти* на клаузата φ.

4.3.14. Пример. Интуитивният смисъл на горната дефиниция се състои в преименуване променливите. Да разгледаме например клаузата

$$\mathtt{p(x,y)}\ \mathtt{:-}\ \mathtt{q(x,x)}, \mathtt{r(z)}$$

Всяка една от следните клаузи е неин вариант:

$$\mathtt{p(y,x)}\ \mathtt{:-}\ \mathtt{q(y,y)}, \mathtt{r(z)}$$
$$\mathtt{p}(\mathtt{x}_1, \mathtt{x}_2)\ \mathtt{:-}\ \mathtt{q}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{r}(\mathtt{x}_3)$$
$$\mathtt{p}(\mathtt{x}_1, \mathtt{y}')\ \mathtt{:-}\ \mathtt{q}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{r}(\mathtt{z}_1)$$

Клаузата

$$\mathtt{p}(\mathtt{x}_1, \mathtt{y}')\ \mathtt{:-}\ \mathtt{q}(\mathtt{x}', \mathtt{x}'), \mathtt{r}(\mathtt{z}_1)$$

не е вариант, защото променливата $\mathtt{x}$ е преименувана някъде като $\mathtt{x}_1$ а другаде като $\mathtt{x}'$. Клаузата

$$\mathtt{p(x,x)}\ \mathtt{:-}\ \mathtt{q(x,x)}, \mathtt{r(x)}$$

също не е вариант, защото е получена посредством субституция v, за която $v(\mathtt{x}) = v(\mathtt{y}) = v(\mathtt{z}) = \mathtt{x}$, а такава субституция не е биективна функция.

Вече може да пристъпим към дефиницията на обратната изводимост с ограничения.

*За целта обаче трябва да поискаме заключенията на клаузите да имат вида $\mathtt{p}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, където $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са не произволни термове, а различни по между си променливи.

✓ **4.3.15. Дефиниция.** Нека Γ е логическа програма и ни е дадено състояние

$$\langle \texttt{?-}\ \texttt{p}(\tau_1, \ldots, \tau_n), \varphi_1, \varphi_2, \ldots, \varphi_k \parallel \psi_1, \ldots, \psi_l \rangle$$

Тогава:

а) Ако клаузата

$$\texttt{p}(\sigma_1, \ldots, \sigma_n)\ \texttt{:-}\ \chi_1, \chi_2, \ldots, \chi_m$$

е вариант на някоя клауза от логическата програма Γ, то даденото състояние *се свежда едностъпково с обратна изводимост* по първи начин при програма Γ до

$$\langle \texttt{?-}\ \chi_1, \chi_2, \ldots, \chi_m, \varphi_1, \varphi_2, \ldots, \varphi_k \parallel \tau_1 = \sigma_1\ \ldots, \tau_n = \sigma_n, \psi_1, \ldots, \psi_l \rangle$$

б) Ако p е символ от ВГРАД, то даденото състояние *се свежда едностъпково с обратна изводимост* по втори начин при програма Γ до

$$\langle \texttt{?-}\ \varphi_1, \varphi_2, \ldots, \varphi_k \parallel \texttt{p}(\tau_1, \ldots, \tau_n), \psi_1, \ldots, \psi_l \rangle$$

С други думи, ако първата атомарна формула от текущото запитване е с вграден предикатен символ, то просто я прехвърляме в ограничението.

4.3.16. *Забележка*: а) Когато предикатният символ p не е от ВГРАД, в общия случай текущото състояние се преобразува по недетерминиран начин. Това е така, защото в програмата може да има много клаузи от вида

$$\texttt{p}(\sigma_1, \ldots, \sigma_n)\ \texttt{:-}\ \chi_1, \chi_2, \ldots, \chi_m$$

По принцип това означава, че при многопроцесорни или многоядрени компютри имаме възможност да изпълняваме паралелно тези алтернативни свеждания на текущото състояние. За съжаление езикът пролог не се възползва от паралелизма на съвременните компютри — при него клаузите се обработват последователно, според реда, в който са написани в програмата.

б) Причината, поради използваме не самите клаузи от Γ, а техни варианти, е следната — променливите, които се срещат в текущото състояние, нямат по смисъл нищо общо с променливите, които са били използвани, когато пишем програмата. Затова, за да не възникнат проблеми, работим не непосредствено с клаузите от програмата, а с варианти на клаузите, в които променливите са преименувани, така че да се различават от променливите в състоянието.

в) Да забележим, че дефиницията е формулирана по такъв начин, че когато ограничението в дадено състояние е неизпълнимо, то тогава всички състояния, към които можем да го сведем, също имат неизпълнимо ограничение. Макар в горната дефиниция това да не е изрично споменато, на практика може да игнорираме състоянията, за чиито ограничения решаващият алгоритъм ни каже, че са неизпълними.

Следващата дефиниция формализира това, което в 4.3.10 изказахме неформално.

✓ **4.3.17. Дефиниция.** а) Казваме, че състоянието $\langle \delta \parallel \varepsilon \rangle$ се свежда към $\langle \delta' \parallel \varepsilon' \rangle$ при програма Γ, ако съществува редица от състояния

$$\langle \delta_0 \parallel \varepsilon_0 \rangle, \langle \delta_1 \parallel \varepsilon_1 \rangle, \ldots, \langle \delta_n \parallel \varepsilon_n \rangle$$

в която първото състояние е $\langle \delta \parallel \varepsilon \rangle$, последното е $\langle \delta' \parallel \varepsilon' \rangle$ и всяко състояние се свежда едностъпково към следващото.

б) Нека е дадена логическа програма Γ. Казваме, че при дадената програма *запитването* $\texttt{?-}\,\varphi_1, \ldots, \varphi_n$ *се извежда с обратен извод с отговор* v, ако v е частична оценка, дефинирана само за променливите, които се срещат в запитването, състоянието

$$\langle \texttt{?-}\,\varphi_1, \ldots, \varphi_n \parallel \top \rangle$$

се свежда при програма Γ към

$$\langle \texttt{?-} \parallel \varepsilon \rangle$$

и ограничението ε от последното състояние е вярно в $\mathbf{B}$ при някаква оценка v', която е разширение на v.

Да илюстрираме с два примера работата на език за логическо програмиране с ограничения.

4.3.18. Пример. Нека универсумът на структурата $\mathbf{B}$ е множество, чиито елементи са хората Адам, Каин и Авел. Нека имаме три вградени символа за константи `адам`, `авел` и `каин`, които се интерпретират в $\mathbf{B}$ по очаквания начин. Нека освен това има и два вградени предикатни символа $=$ и $\neq$, които също се интерпретират в $\mathbf{B}$ по очаквания начин, т.е. съответно като равенство и неравенство. Да разгледаме логическа програма, състояща се от следните три клаузи:

$\texttt{син}(\texttt{авел}, \texttt{адам})$
$\texttt{син}(\texttt{каин}, \texttt{адам})$
$\texttt{брат}(X, Y)$:- $X \neq Y, \texttt{син}(X, Z), \texttt{син}(Y, Z)$

Да видим как тази логическа програма ще обработи запитването

$$\texttt{?- брат}(\texttt{каин}, X)$$

Началното състояние ще бъде

$$\langle \texttt{?- брат}(\texttt{каин}, X) \parallel \top \rangle$$

Третата клауза от програмата е единствената клауза, която може да се приложи към това запитване. Да забележим обаче, че променливата X от нашето запитване няма нищо общо с променливата X от тази клауза. Затова преди да използваме третата клауза, трябва да преименуваме нейните променливи например така:

$\texttt{брат}(X_1, Y_1)$:- $X_1 \neq Y_1, \texttt{син}(X_1, Z_1), \texttt{син}(Y_1, Z_1)$

Прилагайки този вариант на третата клауза към началното състояние, получаваме състоянието

$$\langle \texttt{?-}\ X_1 \neq Y_1, \texttt{син}(X_1, Z_1), \texttt{син}(Y_1, Z_1) \parallel \texttt{каин} = X_1 \,\&\, X = Y_1 \rangle$$

което по втори начин веднага се свежда към

$$\langle \texttt{?- син}(X_1, Z_1), \texttt{син}(Y_1, Z_1) \parallel X_1 \neq Y_1 \,\&\, \texttt{каин} = X_1 \,\&\, X = Y_1 \rangle$$

Ако към това състояние приложим първия факт $\texttt{син}(\texttt{авел}, \texttt{адам})$, ще получим състоянието

$$\langle \texttt{?- син}(Y_1, Z_1) \| X_1 = \texttt{авел} \,\&\, Z_1 = \texttt{адам} \,\&\, X_1 \neq Y_1 \,\&\, \texttt{каин} = X_1 \,\&\, X = Y_1 \rangle$$

Ограничението в това състояние е неизпълнимо (защото стойността на X_1 трябва да е равна едновременно на $\texttt{авел}$ и $\texttt{каин}$), така че повече не се интересуваме от него.

Ако вместо това приложим втория факт $\texttt{син}(\texttt{каин}, \texttt{адам})$, ще получим състоянието

$$\langle \texttt{?- син}(Y_1, Z_1) \| X_1 = \texttt{каин} \,\&\, Z_1 = \texttt{адам} \,\&\, X_1 \neq Y_1 \,\&\, \texttt{каин} = X_1 \,\&\, X = Y_1 \rangle$$

Да опростим това състояние, като елиминираме променливите, за които ограничението дава конкретни стойности. Въпреки че за простота формулирахме дефиниции 4.3.15 и 4.3.17 по начин, който не предвижда да се правят подобни опростявания, от доказателството за коректност

и пълнота, което ще направим, ще се види, че това допустимо. И така, след опростяването получаваме следното състояние:

$$\langle \texttt{?- син}(Y_1, \texttt{адам}) \parallel X = Y_1 \,\&\, \texttt{каин} \neq Y_1 \rangle \tag{37}$$

Ако към това състояние приложим първия факт $\texttt{син}(\texttt{авел}, \texttt{адам})$, ще получим състоянието

$$\langle \texttt{?-} \parallel Y_1 = \texttt{авел} \,\&\, \texttt{адам} = \texttt{адам} \,\&\, X = Y_1 \,\&\, \texttt{каин} \neq Y_1 \rangle$$

Ограничението в така полученото състояние е вярно тогава и само тогава, когато променливите X и Y_1 имат стойност авел. Тъй като единствената променлива в първоначалното запитване бе X, то в този момент компютърът може да спре и да ни даде отговор $X = \texttt{авел}$.

Ако запитаме компютъра дали има и други отговори, то той трябва да пробва състоянието (37) и с втория факт $\texttt{син}(\texttt{каин}, \texttt{авел})$. В този случай това състояние ще се сведе към

$$\langle \texttt{?-} \parallel Y_1 = \texttt{каин} \,\&\, \texttt{адам} = \texttt{адам} \,\&\, X = Y_1 \,\&\, \texttt{каин} \neq Y_1 \rangle$$

където ограничението е неизпълнимо, защото стойността на променливата Y_1 трябва да бъде хем равна, хем различна на каин. Затова в този момент компютърът ще отговори, че няма други отговори.

Забележка: В езика за програмиране пролог предикатът за неравенство се нарича dif. Затова на пролог програмата от току-що разгледания пример изглежда по следния начин:

```
син(авел,адам).
син(каин,адам).
брат(X,Y) :- dif(X,Y), син(X,Z), син(Y,Z).
```

Въпреки че още най-първата версия на пролог разполага с предикат dif, в следващите версии за простота той е премахнат и заменен с предиката \=, който обаче е „дефектен“ и няма да работи както трябва, ако термовете, които сравняваме съдържат променливи. Въпреки че на практика всички съвременни реализации на пролог разполагат с „правилния“ предикат dif, най-хубавите книги за пролог са писани по времето когато такъв предикат не е съществувал. Затова много потребители на пролог не са свикнали да се възползват от предиката dif.

Да видим какво ще се случи, ако в горната програма вместо предиката dif решим да използваме \=. Като приложим третата клауза към началното състояние, получаваме състоянието

$$\langle \texttt{?-}\, X_1 \backslash\texttt{=} Y_1, \texttt{син}(X_1, Z_1), \texttt{син}(Y_1, Z_1) \parallel \texttt{каин} = X_1 \,\&\, X = Y_1 \rangle$$

В този момент ограничението ще каже на компютъра, че стойността на променливата X_1 е `каин`. За стойността на променливата Y_1 обаче нищо не се знае, в частност изглежда възможно и тя също да бъде равна на `каин`. Тъй като има начин променливите X_1 и Y_1 да получат равни стойности, предикатът `\=` ще пропадне и изчислението на програмата ще приключи със съобщение, че няма отговори.

Да повторим: предикатът `\=` може да се използва само ако стойността на двата терма, които сравняваме, вече е известна. За разлика от него, предикатът `dif` може да се използва в произволна ситуация.

4.3.19. Пример. Нека универсумът на структурата $\mathbf{B}$ се състои от реалните числа, двуместните функционални символи `+`, `-`, `*` и `/` се интерпретират съответно като събиране, изваждане, умножение и деление на реални числа и имаме два предикатни символа `=` и `<`, които се интерпретират съответно като „равно“ и „по-малко“. За удобство ще считаме, че освен това в $\mathbf{B}$ са дефинирани символи за константи за реални числа. Например нека символът 1 се интерпретира като числото 1, 5.5 се интерпретира като числото $5\frac{1}{2}$ и т. н.*

При такава структура $\mathbf{B}$, може да дефинираме предикат за факториел по следния начин:

$$\texttt{fact}(0, 1).$$
$$\texttt{fact}(N, N * X) \texttt{ :- } 0 < N, \texttt{fact}(N - 1, X).$$

Нека видим как при горната програма ще се обработи запитването `?-` $\texttt{fact}(3, X)$. Началното състояние ще бъде

$$\langle \texttt{?- fact}(3, X) \parallel \top \rangle$$

Ако се опитаме към това състояние да приложим факта $\texttt{fact}(0, 1)$, трябва да получим състоянието

$$\langle \texttt{?-} \parallel 3 = 0, X = 1 \rangle$$

Ограничението на това състояние е неизпълнимо в $\mathbf{B}$, защото формулата $3 = 0$ е невярна в $\mathbf{B}$, и затова не го използваме.

Ако вместо това използваме втората клауза, то успешно ще сведем първоначалното състояние към следното:

$$\langle \texttt{?-} 0 < N_1, \texttt{fact}(N_1 - 1, X_1) \parallel 3 = N_1, X = N_1 * X_1 \rangle$$

*Един възможен метод за проверка на изпълнимост на ограничения при такава структура $\mathbf{B}$ е да използваме симплекс-метода и алгоритъма на Гаус – Жордан, за да елиминираме линейните равенства и неравенства и да отложим обработката на останалите условия, докато те станат линейни.

Забележете, че сме преименували променливите в клаузата — вместо N имаме N_1 и вместо X имаме X_1.

Ако спазваме точно дефиниции 4.3.15 и 4.3.17, сега би трябвало да приложим някоя клауза към така полученото състояние. На практика обаче всички реализации на логически езици в този момент опростяват състоянието. В нашия случай, ако елиминираме променливата N_1 като я заменим навсякъде с 3, ще получим следното състояние:

$$\langle \texttt{?-}\ 0 < 3 - 1, \texttt{fact}(3 - 1, X_1) \parallel X = 3 * X_1 \rangle$$

Тъй като първата атомарна формула в това състояние е с вграден предикатен символ, то просто я прехвърляме в ограничението:

$$\langle \texttt{?-}\ \texttt{fact}(3 - 1, X_1) \parallel 0 < 3 - 1, X = 3 * X_1 \rangle$$

И отново опростяваме като премахнем от ограничението формулата $0 < 3 - 1$, която е вярна в $\mathbf{B}$:

$$\langle \texttt{?-}\ \texttt{fact}(3 - 1, X_1) \parallel X = 3 * X_1 \rangle$$

Ако към това състояние се опитаме да приложим факта $\texttt{fact}(0, 1)$, трябва да получим състоянието

$$\langle \texttt{?-}\ \parallel 3 - 1 = 0 \,\&\, X_1 = 1, X = 3 * X_1 \rangle$$

Ограничението на това състояние е неизпълнимо в $\mathbf{B}$, защото формулата $3 - 1 = 1$ е невярна в $\mathbf{B}$, и затова не го използваме.

Ако вместо това приложим втората клауза, то успешно ще сведем състоянието до следното:

$$\langle \texttt{?-}\ 0 < N_2, \texttt{fact}(N_2 - 1, X_2) \parallel 3 - 1 = N_2, X_1 = N_2 * X_2, X = 3 * X_1 \rangle$$

И отново опростяваме:

$$\langle \texttt{?-}\ 0 < 3 - 1, \texttt{fact}(3 - 1 - 1, X_2) \parallel X = 3 * 2 * X_2 \rangle$$

Първата атомарна формула в това състояние е с вграден предикатен символ, така че просто я прехвърляме в ограничението:

$$\langle \texttt{?-}\ \texttt{fact}(3 - 1 - 1, X_2) \parallel 0 < 3 - 1, X = 3 * 2 * X_2 \rangle$$

И опростяваме, като махнем тази формула от ограничението, защото тя е вярна в $\mathbf{B}$:

$$\langle \texttt{?-}\ \texttt{fact}(3 - 1 - 1, X_2) \parallel X = 3 * 2 * X_2 \rangle$$

Към това състояние не може да приложим факта $\texttt{fact}(0,1)$, защото, както и по-горе, ще получим състояние с неизпълнимо ограничение. Ако вместо това приложим втората клауза, свеждаме горното състояние до следното:

$$\langle \texttt{?-}\ 0 < N_3, \texttt{fact}(N_3 - 1, X_3) \| 3 - 1 - 1 = N_3, X_2 = N_3 * X_3, X = 3 * 2 * X_2 \rangle$$

Опростяваме ограничението:

$$\langle \texttt{?-}\ 0 < 3 - 1 - 1, \texttt{fact}(3 - 1 - 1 - 1, X_3) \parallel X = 3 * 2 * 1 * X_3 \rangle$$

Първата атомарна формула е с вграден предикатен символ. Прехвърляме я в ограничението, след което забелязваме, че тя е вярна в $\mathbf{B}$ и затова я махаме:

$$\langle \texttt{?-}\ \texttt{fact}(3 - 1 - 1 - 1, X_3) \parallel X = 3 * 2 * 1 * X_3 \rangle$$

Ако към това състояние приложим факта $\texttt{fact}(0,1)$, ще получим състоянието

$$\langle \texttt{?-} \parallel 3 - 1 - 1 - 1 = 0, X_3 = 1, X = 3 * 2 * 1 * X_3 \rangle$$

Опростяваме ограничението:

$$\langle \texttt{?-} \parallel X = 6 \rangle$$

По този начин пресмятането завършва успешно с отговор $X = 6$, което наистина е факториелът на числото 3.

Забележка: Структурата $\mathbf{B}$ от горния пример има аритметичен универсум и аритметични функционални и предикатни символи. Както ще видим по-нататък обаче, стандартната структура на езика пролог не е аритметична, а ербранова. Това означава, че горната логическа програма **не е** коректна програма на пролог.

Традиционният метод за аритметика в пролог използва предикати от типа на `is`, които работят по „дефектен“ начин. Например когато по време на изпълнението на програмата компютърът стигне до

```
M is N + 1
```

това ще доведе до съобщение за грешка, ако в този момент стойността на променливата `N` не е известна. Следователно традиционната аритметика на пролог работи не по начина, който може да очакваме от един логически език за програмиране, а подобно на аритметиката в процедурните езици за програмиране. Използвайки традиционната аритметика на пролог, можем да реализираме предиката за факториел по следния начин:

```
fact(0,1).
fact(N,X) :- 0 < N, N1 is N + 1, fact(N1,X1), X is X1 * N.
```

При такава програма разбира се бихме могли да запитаме компютъра какъв е факториелът на числото 3

$$\texttt{?- fact}(3, X).$$

но ако запитаме факториелът на кое число е равен на 6

$$\texttt{?- fact}(N, 6).$$

ще получим съобщение за грешка.

По-новите версии на пролог коригират този недостатък като добавят предикати, чиито имена започват със символа # и които работят по начина, илюстриран в пример 4.3.19. Използвайки тези нови предикати, можем да дефинираме предикат за факториел по следния начин:

```
fact(0, 1).
fact(N, F) :- N #> 0,
              F #> 0,
              N1 #= N - 1,
              F #= N * F1,
              fact(N1, F1).
```

Този предикат може да се използва не само за да пресмятаме факториела на число, но също и за да задаваме въпроси от типа на има ли число, чийто факториел е 3.

Коректност и пълнота на обратната изводимост с ограничения

4.3.20. Уговорка. За да може формулировките в този подраздел да бъдат по-кратки, нека се уговорим, че всички формули (в частност всички клаузи, запитвания и състояния), всички псевдоформули и всички псевдозапитвания са при сигнатура $\mathfrak{sig}$, която е предикатно обогатяване на ВГРАД.

По принцип сме готови да формулираме теоремите за коректност и пълнота на обратната изводимост с ограничения. В дефиниция 4.3.17 б) се казва какво значи не просто „извежда с обратен извод“ ами „извежда с обратен извод с отговор v“. Ако искаме отговорът да се споменава и в теоремите за коректност и пълнота, ще трябва да разширим дефиницията за удовлетворяване по следния начин:

✓ **4.3.21. Дефиниция.** Нека φ е формула и Γ е множество от клаузи. Ще казваме че формулата φ *се удовлетворява* с отговор v при програма Γ, ако v е частична оценка в $\mathbf{B}$, дефинирана само за променливите от φ и φ е вярна при частичната оценка v във всички структури за $\mathfrak{sig}$, които са едновременно предикатно обогатяване на $\mathbf{B}$ и модел на Γ.

Теоремите за коректност и пълнота на обратната изводимост ще покажат, че едно запитване се извежда с обратен извод с отговор v тогава и само тогав, това запитване се удовлетворява с отговор v. Доказателството на тези теореми ще направим като ги сведем към теоремите за коректност и пълнота за обратната изводимост с частни случаи. Връзката между удовлетворяването на запитване φ с отговор v и удовлетворяването на неговия частен случай при оценка v е ясна:

4.3.22. Твърдение. *Нека δ е запитване, Γ — логическа програма и v е частична оценка в $\mathbf{B}$, дефинирана само за променливите, срещащи се в δ. Запитването δ се удовлетворява при програма Γ с отговор v тогава и само тогава, когато частният случай на δ при оценка v се удовлетворява при програма Γ.*

Доказателство. По същество доказателството представлява просто упражнение върху дефинициите. Случаят когато $\delta = \top$ е очевиден. Нека $\delta = \psi_1 \,\&\, \psi_2 \,\&\, \cdots \,\&\, \psi_n$. Нека частните случаи на $\delta, \psi_1, \psi_2, \ldots, \psi_n$ при оценка v са съответно $\delta', \psi_1', \psi_2', \ldots, \psi_n'$. Тъй като $\delta' = \psi_1' \,\&\, \psi_2' \,\&\, \cdots \,\&\, \psi_n'$, то δ' ще се удовлетворява при програма Γ тогава и само тогава, когато всяка от псевдоформулите ψ_i' се удовлетворява при програма Γ. Това е така тогава и само тогава, когато всяка една от тези псевдоформули е вярна във всяка структура, която е едновременно обогатяване на $\mathbf{B}$ и модел на Γ. Последното е така тогава и само тогава, когато всяка от атомарните формули $\psi_1, \psi_2, \ldots, \psi_n$ е вярна при оценка v във всяка структура, която е едновременно обогатяване на $\mathbf{B}$ и модел на Γ (твърдение 4.2.9). Това пък е така когато запитването δ е вярно при оценка v във всяка структура, която е едновременно обогатяване на $\mathbf{B}$ и модел на Γ. Тъй като v е дефинирана само за променливите, срещащи се в δ, то последното се случва тогава и само тогава, когато запитването δ се удовлетворява при програма Γ с отговор v. ∎

По-сложно се доказва връзката между извеждането на дадено състояние с отговор v и извеждането на частния случай на това състояние при частична оценка v. Най-напред ще установим връзка между едностъпковото свеждане на състояния и едностъпковото свеждане на частни случаи. Едва след това от тук като следствие ще получим нужното.

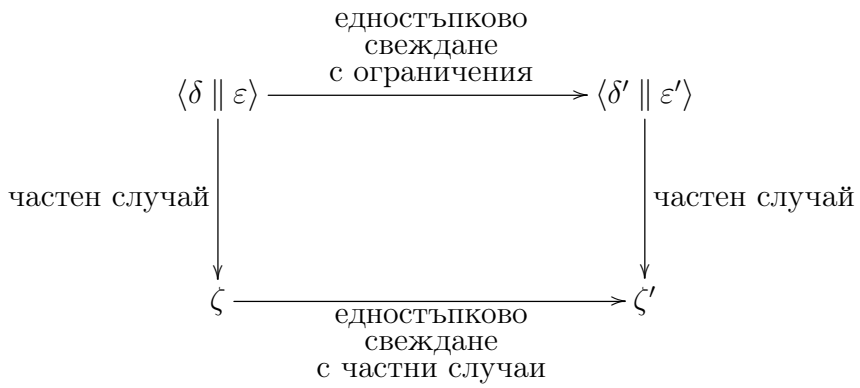
Най-напред да дефинираме какво значи частен случай на състояние.

4.3.23. Дефиниция. а) Нека $\langle \delta \parallel \varepsilon \rangle$ е състояние и v е оценка в $\mathbf{B}$, при която ограничението ε е вярно. В такъв случай частният случай при оценка v на запитването δ е също така *частен случай* при оценка v и на това състояние.

б) Когато v е частична оценка, тогава частните случаи при частичната оценка v на дадено състояние са частните случаи на това състояние при коя да е оценка v', която е обогатяване на v.

Да забележим, че когато v е оценка, едно състояние има не повече от един частен случай при тази оценка. Когато обаче v е частична оценка, тогава е възможно едно състояние да има много частни случаи при тази частична оценка. Това е така, защото в този случай на променливите, за които v не е дефинирана, може да даваме произволни стойности.

Идеята на следващите две леми е съвсем проста, макар формулировката им да е малко тромава, а доказателството — още по-тромаво. Те казват, че ако имаме състояние $\langle \delta \parallel \varepsilon \rangle$, то ще бъде все едно дали най-напред ще сведем това състояние едностъпково до друго състояние $\langle \delta' \parallel \varepsilon' \rangle$ и след това ще вземем частен случай ζ' на така полученото ново състояние, или най-напред ще вземем частен случай ζ на $\langle \delta \parallel \varepsilon \rangle$ и след това ще сведем едностъпково ζ към ζ'.



4.3.24. Лема за повдигането. *Нека* Γ *е логическа програма,* $\langle \delta \parallel \varepsilon \rangle$ *е състояние и* v *е частична оценка в* $\mathbf{B}$*, която е дефинирана само за краен брой променливи. Ако някой частен случай при частичната оценка* v *на* $\langle \delta \parallel \varepsilon \rangle$ *се свежда едностъпково при обратна изводимост с частни случаи към псевдозапитването* ζ'*, то* ζ' *е частен случай при частичната оценка* v *на състояние, към което* $\langle \delta \parallel \varepsilon \rangle$ *се свежда с обратна изводимост при програма* Γ*.*

Доказателство. Да допуснем, че състоянието $\langle \delta \parallel \varepsilon \rangle$ има частен случай ζ при частичната оценка v и ζ се свежда едностъпково с обратна изводимост с частни случаи към ζ'. Трябва да докажем, че състоянието $\langle \delta \parallel \varepsilon \rangle$ се свежда едностъпково с обратна изводимост при програма Γ към такова състояние $\langle \delta' \| \varepsilon' \rangle$, че ζ' е негов частен случай при частичната оценка v.

Нека състоянието $\langle \delta \parallel \varepsilon \rangle$ има вида

$$\langle \texttt{?- p}(\tau_1, \ldots, \tau_n), \varphi_1, \ldots, \varphi_k \parallel \varepsilon \rangle \tag{38}$$

Нека $\varphi'_1, \ldots, \varphi'_k$ са частните случаи при частична оценка v съответно на $\varphi_1, \ldots, \varphi_k$ нека при тази оценка термовете $\tau_1, \ldots, \tau_n$ имат стойности съответно $\alpha_1, \ldots, \alpha_n$. В такъв случай псевдозапитването ζ ще има вида

$$\texttt{?- p}(\alpha_1, \ldots, \alpha_n), \varphi'_1, \ldots, \varphi'_k \tag{39}$$

Ще разгледаме два случая според това дали предикатният символ $\texttt{p}$ е вграден или не. Когато той не е вграден, щом ζ се свежда едностъпково към ζ', то значи някоя клауза от Γ

$$\texttt{p}(\sigma_1, \ldots, \sigma_n) \texttt{ :- } \chi_1, \ldots, \chi_m \tag{40}$$

има частен случай от вида

$$\texttt{p}(\alpha_1, \ldots, \alpha_n) \texttt{ :- } \chi'_1, \ldots, \chi'_m \tag{41}$$

и псевдозапитването ζ' има вида

$$\texttt{?- } \chi'_1, \ldots, \chi'_m, \varphi'_1, \ldots, \varphi'_k \tag{42}$$

Нека

$$\texttt{p}(\sigma'_1, \ldots, \sigma'_n) \texttt{ :- } \chi''_1, \ldots, \chi''_m \tag{43}$$

е вариант на клауза (40), който не съдържа променливи, за които частичната оценка v е дефинирана (такъв вариант има, защото v е дефинирана само за краен брой променливи). Нека w е оценка, която на всяка променлива от (43) дава същата стойност, каквато частичната оценка v дава на съответната променлива от (40). Да дефинираме оценката v' по следния начин:

$$v'(\texttt{x}) = \begin{cases} v(\texttt{x}), & \text{ако } v \text{ е дефинирана за } \texttt{x} \\ w(\texttt{x}), & \text{ако } \texttt{x} \text{ е променлива от (43)} \\ \text{произволно}, & \text{иначе} \end{cases}$$

Така дефинираната оценка v' е обогатяване на v, а частният случай на (43) при оценка v' е псевдоклауза (41).

Използвайки варианта (43) на клауза (40), състоянието $\langle \delta \parallel \varepsilon \rangle$ (38) се свежда едностъпково (по първи начин) до състоянието

$$\langle \texttt{?-}\, \chi''_1, \ldots, \chi''_m, \varphi_1, \ldots, \varphi_k \parallel \tau_1 = \sigma'_1, \ldots, \tau_n = \sigma'_n, \varepsilon \rangle$$

Ограничението на това състояние е вярно при оценка v', защото термовете τ_i и σ'_i имат една и съща стойност (равна на α_i), а ε е вярно при оценка v', защото за променливите, срещащи се в ε оценката v' съвпада с частичната оценка v, а ε е част от състоянието $\langle \delta \parallel \varepsilon \rangle$, което има частен случай при частичната оценка v. Щом като това ограничението на това състояние е вярно при оценка v', то значи то има частен случай при тази оценка. От начина, по който дефинирахме оценката v' може да се съобрази, че този частен случай е ζ' (42).

Сега да разгледаме случая, когато предикатният символ $\mathtt{p}$ е вграден. Този случай е по-лесен. Щом псевдозапитването ζ (39) се свежда едностъпково към ζ', то значи първата псевдоформула на ζ, а именно $\mathtt{p}(\alpha_1, \ldots, \alpha_n)$ е вярна в $\mathbf{B}$ и псевдозапитването ζ' има вида

$$\texttt{?-}\, \varphi'_1, \ldots, \varphi'_k \tag{44}$$

От това че псевдоформулата $\mathtt{p}(\alpha_1, \ldots, \alpha_n)$ е вярна в $\mathbf{B}$ следва, че формулата $\mathtt{p}(\tau_1, \ldots, \tau_n)$ е вярна в $\mathbf{B}$ при частична оценка v.

Състоянието $\langle \delta \parallel \varepsilon \rangle$ (38) се свежда по втори начин към състоянието

$$\langle \texttt{?-}\, \varphi_1, \ldots, \varphi_k \parallel \mathtt{p}(\tau_1, \ldots, \tau_n), \varepsilon \rangle \tag{45}$$

Тъй като вече установихме, че формулата $\mathtt{p}(\tau_1, \ldots, \tau_n)$ е вярна при частичната оценка v, а за ε вече знаем това, защото иначе състоянието $\langle \delta \parallel \varepsilon \rangle$ не би имало частен случай при тази частична оценка, то значи състоянието (45) има частен случай при частичната оценка v. Този частен е точно псевдозапитването ζ' (44). ∎

4.3.25. Лема за спускането. *Нека* Γ *е логическа програма,* $\langle \delta \parallel \varepsilon \rangle$ *е състояние и* v *е частична оценка в* $\mathbf{B}$*, която е дефинирана само за краен брой променливи. Ако псевдозапитването* ζ' *е частен случай при частичната оценка* v *на състояние, към което* $\langle \delta \parallel \varepsilon \rangle$ *се свежда с обратна изводимост при програма* Γ*, то някой частен случай при частичната оценка* v *на* $\langle \delta \parallel \varepsilon \rangle$ *се свежда едностъпково при обратна изводимост с частни случаи към* ζ'*.*

Доказателство. Да допуснем, че състоянието $\langle \delta \,\|\, \varepsilon \rangle$ се свежда едностъпково с обратна изводимост при програма Г към $\langle \delta' \parallel \varepsilon' \rangle$ и ζ' е частен случай при частичната оценка v на $\langle \delta' \parallel \varepsilon' \rangle$. Трябва да докажем, че състоянието $\langle \delta \parallel \varepsilon \rangle$ има такъв частен случай ζ при частичната оценка v, че ζ се свежда едностъпково с обратна изводимост с частни случаи към ζ'.

Ако предикатният символ на първата формула от запитването δ не е вграден (т.е. ако свеждането на $\langle \delta \parallel \varepsilon \rangle$ към $\langle \delta' \parallel \varepsilon' \rangle$ е по първи начин, вж. дефиниция 4.3.15), тогава $\langle \delta \parallel \varepsilon \rangle$ ще има вида

$$\langle \texttt{?-}\, \mathtt{p}(\tau_1, \ldots, \tau_n), \varphi_1, \ldots, \varphi_k \parallel \varepsilon \rangle$$

някоя клауза от Г ще има вариант

$$\mathtt{p}(\sigma_1, \ldots, \sigma_n) \texttt{:-}\, \chi_1, \ldots, \chi_m \tag{46}$$

и състоянието $\langle \delta' \parallel \varepsilon' \rangle$ ще бъде равно на

$$\langle \texttt{?-}\, \chi_1, \ldots, \chi_m, \varphi_1, \ldots, \varphi_k \parallel \tau_1 = \sigma_1 \ldots, \tau_n = \sigma_n, \varepsilon \rangle \tag{47}$$

Ако v' е оценка, която обогатява частичната оценка v и при която е получен частният случай ζ' на $\langle \delta' \parallel \varepsilon' \rangle$, то ограничението от горното състояние трябва да бъде вярно при оценка v' и освен това псевдозапитването ζ' има вида

$$\texttt{?-}\, \chi'_1, \ldots, \chi'_m, \varphi'_1, \ldots, \varphi'_k \tag{48}$$

където $\chi'_1, \ldots, \chi'_m, \varphi'_1, \ldots, \varphi'_k$ са частните случаи при оценка v' съответно на $\chi_1, \ldots, \chi_m, \varphi_1, \ldots, \varphi_k$

Да положим ζ да бъде частният случай при оценка v' на $\langle \delta \| \varepsilon \rangle$; такъв частен случай съществува, защото ограничението на (47) е вярно при оценка v', а ε е част от това ограничение. Нека още $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ са стойностите в $\mathbf{B}$ при оценка v' съответно на $\tau_1, \ldots, \tau_n, \sigma_1, \ldots, \sigma_n$. В такъв случай псевдозапитването ζ ще има вида

$$\texttt{?-}\, \mathtt{p}(\alpha_1, \ldots, \alpha_n), \varphi'_1, \ldots, \varphi'_k \tag{49}$$

Тъй като ограничението на състояние (47) е вярно при оценка v', то $\alpha_1 = \beta_1, \ldots, \alpha_n = \beta_n$ и значи с помощта на псевдоклаузата

$$\mathtt{p}(\beta_1, \ldots, \beta_n) \texttt{:-}\, \chi'_1, \ldots, \chi'_m$$

която е частен случай при оценка v' на клауза (46), псевдозапитването ζ (49) се свежда едностъпково с обратен извод с частни случаи към ζ' (48).

Ако свеждането на $\langle \delta \parallel \varepsilon \rangle$ към $\langle \delta' \parallel \varepsilon' \rangle$ е по втори начин (вж. дефиниция 4.3.15), тогава $\langle \delta \parallel \varepsilon \rangle$ ще има вида

$$\langle \texttt{?-}\, \mathtt{p}(\tau_1, \ldots, \tau_n), \varphi_1, \ldots, \varphi_k \parallel \varepsilon \rangle$$

и $\langle \delta' \parallel \varepsilon' \rangle$ ще има вида

$$\langle \texttt{?-}\, \varphi_1, \ldots, \varphi_k \parallel \mathtt{p}(\tau_1, \ldots, \tau_n), \varepsilon \rangle \tag{50}$$

Ако v' е оценка, която обогатява частичната оценка v и при която е получен частният случай ζ' на $\langle \delta' \parallel \varepsilon' \rangle$, то ограничението от горното състояние трябва да бъде вярно при оценка v' и освен това псевдозапитването ζ' ще има вида

$$\texttt{?-}\, \varphi'_1, \ldots, \varphi'_k \tag{51}$$

където $\varphi'_1, \ldots, \varphi'_k$ са частните случаи при оценка v' съответно на $\varphi_1, \ldots, \varphi_k$.

Да положим ζ да бъде частният случай при оценка v' на $\langle \delta \parallel \varepsilon \rangle$; такъв частен случай съществува, защото ограничението на (50) е вярно при оценка v', а ε е част от това ограничение. Нека още $\alpha_1, \ldots, \alpha_n$ са стойностите в $\mathbf{B}$ при оценка v' съответно на $\tau_1, \ldots, \tau_n$. В такъв случай псевдозапитването ζ има вида

$$\texttt{?-}\, \mathtt{p}(\alpha_1, \ldots, \alpha_n), \varphi'_1, \ldots, \varphi'_k \tag{52}$$

За да видим, че ζ (52) се свежда с обратна изводимост с частни случаи към ζ' (51), остава само да забележим, че псевдоформулата $\mathtt{p}(\alpha_1, \ldots, \alpha_n)$ е вярна в $\mathbf{B}$, защото формулата $\mathtt{p}(\tau_1, \ldots, \tau_n)$ е част от ограничението на (50) и значи е вярна в $\mathbf{B}$ при оценка v'. ∎

4.3.26. Следствие за спускането и повдигането. *Нека* Γ *е логическа програма,* δ *е запитване и* v *е частична оценка, дефинирана само за променливите, срещащи се в* δ*. Запитването* δ *се извежда с обратен извод с отговор* v *при програма* Γ *тогава и само тогава, когато* δ *има частен случай при частичната оценка* v *и той се извежда с обратна изводимост с частни случаи.*

Доказателство. ($\Longrightarrow$) Трябва да докажем, че ако δ се извежда с обратен извод с отговор v, то δ има частен случай при частичната оценка v, който се извежда с обратна изводимост с частни случаи.

Щом δ се извежда с обратен извод с отговор v, то значи имаме такава редица

$$\langle \delta_0 \parallel \varepsilon_0 \rangle \mapsto \langle \delta_1 \parallel \varepsilon_1 \rangle \mapsto \langle \delta_2 \parallel \varepsilon_2 \rangle \mapsto \cdots \mapsto \langle \delta_n \parallel \varepsilon_n \rangle$$

че всяко състояние в нея се свежда едностъпково към следващото, $\delta_0 = \delta$, $\delta_n = \top$ и ограничението ε_n е вярно при някаква оценка v', която е обогатяване на v.

Нека ζ_n е частният случай на $\langle \delta_n \parallel \varepsilon_n \rangle$ при оценка v' (такъв частен случай има, защото ограничението ε_n вярно при оценка v'). Тъй като $\delta_n = \top$, то $\zeta_n = \top$. Разбира се, щом v' е обогатяване на v и ζ_n е частен случай при оценка v', то ζ_n е също така частен случай и при частичната оценка v.

Съгласно лема 4.3.25 състоянието $\langle \delta_{n-1} \parallel \varepsilon_{n-1} \rangle$ има такъв частен случай ζ_{n-1} при частичната оценка v, че ζ_{n-1} се свежда едностъпково към ζ_n. Пак от предната лема следва, че състоянието $\langle \delta_{n-2} \parallel \varepsilon_{n-2} \rangle$ има такъв частен случай ζ_{n-2} при частичната оценка v, че ζ_{n-2} се свежда едностъпково към ζ_{n-1}. Още едно прилагане на предната лема ни дава такъв частен случай ζ_{n-3} при частичната оценка v на състоянието $\langle \delta_{n-3} \parallel \varepsilon_{n-3} \rangle$, че ζ_{n-3} се свежда едностъпково към ζ_{n-2}. Продължавайки по този начин, намираме частен случай ζ_0 на първото състояние $\langle \delta_0 \| \varepsilon_0 \rangle$, който се свежда едностъпково към ζ_1.

Щом ζ_0 се свежда едностъпково към ζ_1, ζ_1 към ζ_2, ζ_2 към ζ_3,..., ζ_{n-1} към ζ_n, и $\zeta_n = \top$, то значи ζ_0 се извежда с обратна изводимост с частни случаи.

($\Longleftarrow$) Трябва да докажем, че ако δ има частен случай при частичната оценка v, който се извежда с обратна изводимост с частни случаи, то δ се извежда с обратна изводимост с отговор v.

Нека ζ_0 е частният случай при частичната оценка v на δ, който се извежда с обратна изводимост с частни случаи. Нека $\delta_0 = \delta$ и $\varepsilon_0 = \top$. Тъй като ограничението ε_0 е вярно при коя да е оценка, то ζ_0 е частен случай също и на състоянието $\langle \delta_0 \parallel \varepsilon_0 \rangle$. Освен това, щом ζ_0 се извежда с обратна изводимост, то значи имаме редица

$$\zeta_0 \mapsto \zeta_1 \mapsto \zeta_2 \mapsto \cdots \mapsto \zeta_n$$

в която всяко псевдозапитване се свежда към следващото и $\zeta_n = \top$.

Щом ζ_0 се свежда едностъпково към ζ_1, то от лема 4.3.24 следва, че има такова състояние $\langle \delta_1 \parallel \varepsilon_1 \rangle$, че ζ_1 е негов частен случай при частичната оценка v и $\langle \delta_0 \| \varepsilon_0 \rangle$ се свежда едностъпково към $\langle \delta_1 \| \varepsilon_1 \rangle$. Щом ζ_1 се свежда едностъпково към ζ_2, то от предната лема следва, че има такова състояние $\langle \delta_2 \| \varepsilon_2 \rangle$, че ζ_2 е негов частен случай при частичната оценка v и $\langle \delta_1 \| \varepsilon_1 \rangle$ се свежда едностъпково към $\langle \delta_2 \| \varepsilon_2 \rangle$. Продължавайки по този начин намираме редица от състояния

$$\langle \delta_0 \parallel \varepsilon_0 \rangle \mapsto \langle \delta_1 \parallel \varepsilon_1 \rangle \mapsto \langle \delta_2 \parallel \varepsilon_2 \rangle \mapsto \cdots \mapsto \langle \delta_n \parallel \varepsilon_n \rangle$$

в която всяко състояние се свежда едностъпково към следващото и псевдозапитването ζ_n е частен случай при частичната оценка v на състоянието $\langle \delta_n \parallel \varepsilon_n \rangle$. Следователно съществува оценка v', която е обогатяване на v, при която този частен случай се получава. Ограничението ε_n, разбира се, трябва да бъде вярно при тази оценка. Остава само да забележим, че от $\zeta_n = \top$ следва $\delta_n = \top$ и значи δ_0 се извежда с обратна изводимост с отговор v. ∎

4.3.27. Теорема за коректност на обратната изводимост с ограничения. *Ако запитването δ се извежда с обратен извод при програма Γ с отговор v, то δ се удовлетворява при програма Γ с отговор v.*

Доказателство. Току-що доказаното следствие показва, че δ има частен случай ζ при частичната оценка v, който се извежда с обратна изводимост с частни случаи. От теоремата за коректност на обратната изводимост с частни случаи следва, че ζ се удовлетворява при програма Γ. От твърдение 4.3.22 следва, че δ се удовлетворява при програма Γ с отговор v. ∎

4.3.28. Теорема за пълнота на обратната изводимост с ограничения. *Ако запитването δ се удовлетворява при програма Γ с отговор v, то δ се извежда с обратен извод при програма Γ с отговор v.*

Доказателство. От твърдение 4.3.22 следва, че δ има частен случай ζ при частичната оценка v, който се удовлетворява при програма Γ. От теоремата за пълнота на обратната изводимост с частни случаи следва, че ζ се извежда с обратна изводимост. От по-горе доказаното следствие следва, че δ се извежда с обратна изводимост с отговор v. ∎

4.4. Ербранови структури

В предните раздели се уговорихме, че разполагаме с фиксирана структура **B**. Тази структура определя какви вградени функции и предикати има в езика за логическо програмиране, а от нейния универсум разбираме с какви обекти може да работим в програмата. За логическата програма може да кажем, че надгражда над структурата **B**, т.е. приемаме, че имаме наготово всичко, предоставено от **B** и използваме логика, за да дефинираме нови предикати. Затова почти всички дефиниции в предните раздели споменаваха структурата **B**. Така например за едно запитване φ казахме, че се удовлетворява при дадена

логическа програма с отговор v, ако φ е вярна при оценка v във всички структури, които не само бяха модел на програмата, но също са и обогатявания на $\mathbf{B}$ (вж. дефиниция 4.3.21).

Въпреки че когато програмираме този подход е естествен, от логическа гледна точка е по-естествено да разгледаме вариант на логическото програмиране, при който няма структура $\mathbf{B}$. С други думи, интересен е вариант на логическото програмиране, при който не използваме логиката, за да надграждаме над нещата, предоставени от $\mathbf{B}$, а абсолютно всичко свеждаме към логика. Можем да дадем следната дефиниция:

4.4.1. Дефиниция. За едно запитване φ казваме, че се удовлетворява при програма Г, ако запитването φ е изпълнимо във всеки модел на Г.

Следователно за едно запитване φ казваме, че се удовлетворява, ако φ е изпълнима във всички модели на логическата програма, а не само при моделите, които са обогатявания на $\mathbf{B}$. Разбира се, при този подход няма как да кажем с какви отговори се удовлетворява запитването, защото конкретните отговори зависят от универсума на структурата, а нямаме структура $\mathbf{B}$, която да ни каже кой е универсумът.

За щастие, оказва се, че не се налага да разработваме нов вид теория на логическото програмиране. Френският логик Жак Ербран (1908 – 1931) е установил, че вместо да разглеждаме произволни структури с кой знае какъв универсум, е достатъчно да се ограничим със структури, чийто универсум се състои от всички термове без променливи, а стойността на кой да е терм без променливи в тази структура е самият терм. Такива структури се наричат *ербранови*.

Нека отбележим, че ербрановите структури не винаги съществуват. Съгласно задача 15, ако в сигнатурата няма нито един символ за константи, то няма да съществуват термове без променливи, а универсумите на структурите не е позволено да бъдат празното множество. Ако пък в сигнатурата има поне един символ за константа, то този символ сам си е терм без променливи, така че в този случай ербранови структури ще съществуват.

4.4.2. Всичко в този раздел ще правим при предположението, че в сигнатурата има поне един символ за константи. На практика ограничението да имаме поне един символ за константа изобщо не е ограничително. Дори и да нямаме такъв символ, можем да променим сигнатурата като добавим в нея някакъв символ за константа, защото тази промяна няма да доведе до никакви съществени изменения.

✓ **4.4.3. Дефиниция.** Структурата $\mathbf{H}$ е *ербранова*, ако:

- Универсумът на $\mathbf{H}$ е множеството от всички термове, които не съдържат променливи.
- За всеки символ за константа $\mathtt{c}$, стойността му е самата константа, т.е. $\mathtt{c}^{\mathbf{H}} = \mathtt{c}$. (Да забележим, че $\mathtt{c}$ е терм без променливи.)
- За всеки n-местен функционален символ $\mathtt{f}$ и елементи $\tau_1, \tau_2, \ldots, \tau_n$ на универсума на $\mathbf{H}$ имаме

$$\mathtt{f}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n) \tag{53}$$

(Да забележим, че щом $\tau_1, \tau_2, \ldots, \tau_n$ са от универсума на $\mathbf{H}$, то те са термове без променливи, а значи и $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ е терм без променливи, т.е. елемент на универсума.)

4.4.4. Дефиниция. Множеството от всички термове без променливи ще наричаме *ербранов универсум*.

4.4.5. Обърнете внимание, че скобите от двете страни на равенство (53) имат напълно различен смисъл. Скобите отляво ги използваме, за да покажем, че даваме $\tau_1, \tau_2, \ldots, \tau_n$ като аргументи на функцията $\mathtt{f}^{\mathbf{H}}$. Скобите отдясно пък са просто символи — вторият и последният символ на терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Също така напълно различен е и смисълът на запетаите. Използваме запетаите отляво, за да разделим аргументите на функцията $\mathtt{f}^{\mathbf{H}}$. За разлика от тях, запетаите отдясно са просто символи, които се срещат някъде в терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$.

***Задача* 48:** Нека сигнатурата съдържа поне един символ за константа. Докажете подробно, че съществуват ербранови структури.

Тъй като елементите на универсума на една ербранова структура са термове без променливи, то всяка оценка в тази структура съпоставя на променливите термове без променливи. Това означава, че всяка оценка в ербранова структура представлява субституция и значи можем да я прилагаме по два различни начина:

- можем да пресметнем стойността на един терм при тази оценка;
- можем да си мислим, че оценката е субституция и да я приложим към този терм.

Следващото твърдение показва, че и в двата случая ще получим един и същ резултат, т.е. $[\![\tau]\!]^{\mathbf{H}} v = \tau v$.

✓ **4.4.6. Твърдение.** *Нека* $\mathbf{H}$ *е ербранова структура и* v *е оценка в* $\mathbf{H}$. *Тогава стойността на кой да е терм* τ *в* $\mathbf{H}$ *при оценка* v *е равна на резултата от прилагането на субституцията* v *към терма* τ.

✓ Доказателство. Ще докажем твърдението с индукция по терма τ. Да припомним, че с $[\![\tau]\!]^{\mathbf{H}}v$ означаваме стоността на τ при оценка v в структурата $\mathbf{H}$, а τv е резултатът от прилагането на субституцията v към τ.

Ако $\tau = \mathtt{x}$ е променлива, то стойността на τ при оценка v е стойността на $\mathtt{x}$ при оценка v, т.е. $v(\mathtt{x})$. Да приложим субституцията v към $\mathtt{x}$ означава да заместим $\mathtt{x}$ с $v(\mathtt{x})$ и значи отново получаваме $v(\mathtt{x})$.

Ако $\tau = \mathtt{c}$ е символ за константа, то по дефиниция 3.2.2 б) стойността на τ в $\mathbf{H}$ при коя да е оценка е $\mathtt{c}^{\mathbf{H}} = \mathtt{c}$. Резултатът от прилагането на коя да е субституция към терма $\mathtt{c}$ също е $\mathtt{c}$.

Нека $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Стойността на τ в $\mathbf{H}$ при оценка v е

$$[\![\tau]\!]^{\mathbf{H}}v = \mathtt{f}^{\mathbf{H}}([\![\tau_2]\!]^{\mathbf{H}}v, [\![\tau_2]\!]^{\mathbf{H}}v, \ldots, [\![\tau_n]\!]^{\mathbf{H}}v)$$

Понеже структурата е ербранова, последното е равно на

$$\mathtt{f}([\![\tau_1]\!]^{\mathbf{H}}v, [\![\tau_2]\!]^{\mathbf{H}}v, \ldots, [\![\tau_n]\!]^{\mathbf{H}}v)$$

което съгласно индукционното предположение е равно на

$$\mathtt{f}(\tau_1 v, \tau_2 v, \ldots, \tau_n v) = (\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n))v = \tau v$$

■

✓ **4.4.7. Следствие.** *Ако* $\mathbf{H}$ *е ербранова структура и* τ *е терм без променливи, то стойността на* τ *в* $\mathbf{H}$ *при коя да е оценка е* τ*.*

✓ Доказателство. Нека v е произволна оценка в $\mathbf{H}$. Съгласно твърдение 4.4.6 стойността на τ в $\mathbf{H}$ при оценка v е равна на резултата от прилагането на субституцията v към τ. Но τ не съдържа променливи, които субституциите могат да заместват и значи като приложим v към τ получаваме пак τ. ■

✓ ***Задача* 49:** Нека $\mathbf{H}$ е ербранова структура и оценката v в $\mathbf{H}$ е такава, че $v(\mathtt{x}) = \mathtt{f}(\mathtt{f}(\mathtt{c}))$ и $v(\mathtt{y}) = \mathtt{g}(\mathtt{c}, \mathtt{f}(\mathtt{a}))$. Кои от скобите и кои от запетаите в следния израз са символи и кои не са символи:

$$\mathtt{f}(\mathtt{g}^{\mathbf{H}}(v(\mathtt{x}), (\mathtt{g}(\mathtt{a}, \mathtt{y}))v))$$

Колко леви скоби и колко запетаи се съдържат в стойността на този израз?

За формулите вместо твърдение 4.4.6 можем да докажем следното твърдение:

✓ **4.4.8. Твърдение.** *Нека* $\mathbf{H}$ *е ербранова* $\mathfrak{sig}$*-структура и* v *е оценка в* $\mathbf{H}$*. Една* $\mathfrak{sig}$*-формула* φ *е вярна в* $\mathbf{H}$ *при оценка* v *тогава и само тогава, когато формулата* φv *е вярна в* $\mathbf{H}$*.*

Доказателство. Нека $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са свободните променливи във φ.

Нека w е произволна оценка в $\mathbf{H}$. Съгласно лемата за субституциите 3.3.17,

$$\begin{aligned}\mathbf{H} \vDash \varphi v[w] &\longleftrightarrow \mathbf{H} \vDash \varphi[\mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq v(\mathtt{x}_1), \ldots, v(\mathtt{x}_n)][w] \\ &\longleftrightarrow \mathbf{H} \vDash \varphi[\mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq [\![v(\mathtt{x}_1)]\!]^{\mathbf{H}} w, \ldots, [\![v(\mathtt{x}_n)]\!]^{\mathbf{H}} w]\end{aligned}$$

Тъй като термовете $v(\mathtt{x}_1), v(\mathtt{x}_2), \ldots, v(\mathtt{x}_n)$ не съдържат променливи, съгласно следствие 4.4.7, $[\![v(\mathtt{x}_i)]\!]^{\mathbf{H}} w = v(\mathtt{x}_i)$ и значи може да довършим горната редица от еквивалентности по следния начин:

$$\begin{aligned}\mathbf{H} \vDash \varphi[\mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq &[\![v(\mathtt{x}_1)]\!]^{\mathbf{H}} w, \ldots, [\![v(\mathtt{x}_n)]\!]^{\mathbf{H}} w] \longleftrightarrow \\ &\longleftrightarrow \mathbf{H} \vDash \varphi[\mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq v(\mathtt{x}_1), \ldots, v(\mathtt{x}_n)] \\ &\longleftrightarrow \mathbf{H} \vDash \varphi[v]\end{aligned}$$

∎

4.4.9. Лема. *Нека* $\mathbf{M}$ *е произволна структура. Тогава съществува ербранова структура* $\mathbf{H}$ *и силен хомоморфизъм* $h\colon \mathbf{H} \to \mathbf{M}$*.*

Доказателство. Дефиниция 4.4.3 за ербранова структура ни казва какъв е универсумът на $\mathbf{H}$ — множеството от всички термове, които не съдържат променливи. Пак от дефиницията разбираме и как в $\mathbf{H}$ се интерпретират символите за константи и функционалните символи. Остава единствено да определим как в $\mathbf{H}$ се интерпретират предикатните символи. Да отложим засега това.

Хомоморфизмът h трябва да изобразява всеки терм без променливи в елемент на универсума на $\mathbf{M}$. Да дефинираме $h(\tau)$ да бъде стойността на τ в $\mathbf{M}$ (тъй като τ не съдържа променливи, не е нужно да уточняваме коя е оценката, вж. твърдение 3.2.5). Да докажем, че това наистина е силен хомоморфизъм.

Първо, по дефиниция h изобразява елементите на универсума на $\mathbf{H}$ е елементи на универсума на $\mathbf{M}$.

Второ, за всеки символ за константа $\mathtt{c}$:

$$h(\mathtt{c}^{\mathbf{H}}) = h(\mathtt{c}) = \mathtt{c}$$

Освен това, за всеки n-местен функционален символ $\mathtt{f}$ (в долните равенства с $[\![\ldots]\!]^{\mathbf{M}}$ е означена стойността в $\mathbf{M}$ на терм без променливи):

$$\begin{aligned} h(\mathtt{f}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n)) &= h(\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)) \\ &= [\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)]\!]^{\mathbf{M}} \\ &= \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}, [\![\tau_2]\!]^{\mathbf{M}}, \ldots, [\![\tau_n]\!]^{\mathbf{M}}) \\ &= \mathtt{f}^{\mathbf{M}}(h(\tau_1), h(\tau_2), \ldots, h(\tau_n)) \end{aligned}$$

И накрая, за всеки n-местен предикатен символ $\mathtt{p}$ може просто да дефинираме

$$\mathtt{p}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) \overset{\text{def}}{\longleftrightarrow} \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}, [\![\tau_2]\!]^{\mathbf{M}}, \ldots, [\![\tau_n]\!]^{\mathbf{M}})$$

защото тогава

$$\mathtt{p}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) \longleftrightarrow \mathtt{p}^{\mathbf{M}}(h(\tau_1), h(\tau_2), \ldots, h(\tau_n))$$

■

✓ **4.4.10. Теорема („малка“ теорема на Ербран).** *а) Нека* Γ *е множество от безквантори формули и* φ *е безквантора формула. Тогава* φ *е изпълнима във всеки модел на* Γ *тогава и само тогава, когато* φ *е изпълнима във всеки ербранов модел на* Γ.

б) Множество Γ *от безквантори формули има модел тогава и само тогава, когато то има ербранов модел.*

Доказателство. **(а)** Ясно е, че ако φ е изпълнима във всеки модел на Γ, то в частност φ ще бъде изпълнима и във всеки ербранов модел на Γ. За да докажем обратната посока, да допуснем, че φ е изпълнима във всеки ербранов модел на Γ и да изберем произволен (не задължително ербранов) модел на Γ. Трябва да докажем, че φ е изпълнима в $\mathbf{M}$.

От лема 4.4.9 намираме ербранова структура $\mathbf{H}$ и хомоморфизъм $h\colon \mathbf{H} \to \mathbf{M}$. От твърдение 3.5.9 а) следва, че $\mathbf{H}$ е модел на Γ. Тъй като φ е изпълнима във всеки ербранов модел на Γ, то значи φ е изпълнима в $\mathbf{H}$, следователно от твърдение 3.5.9 б) получаваме, че φ е изпълнима в $\mathbf{M}$.

(б) Може да сведем към (а). Нека $\varphi = \bot$. Тъй като φ е неизпълнима формула, то φ е изпълнима във всеки модел на Γ тогава и само тогава, когато Γ няма модели и φ е изпълнима във всеки ербранов модел на Γ тогава и само тогава, когато Γ няма ербранови модели. ■

4.4.11. Може да използваме „малката“ теорема на Ербран, за да сведем логическото програмиране без структура **B** към логическото програмиране със структура **B**. Нека единственият предикатен символ в сигнатурата ВГРАД е равенството. И нека **B** е ербрановата структура за тази сигнатура, в която символът за равенство се интерпретира като равенство (т.е. универсумът на **B** съдържа термовете без променливи, а символите за константи и функционалните символи се интерпретират съгласно дефиниция 4.4.3). При така дефинирана структура **B** е вярно следното следствие:

✓ **4.4.12. Следствие.** *Нека* Г *е логическа програма, която не съдържа символа за равенство и* φ *е запитване, което също не съдържа символа за равенство. Тогава* φ *се удовлетворява от* Г *(вж. дефиниция 4.4.1) тогава и само тогава, когато съществува отговор, с който при тази програма запитването* φ *се удовлетворява над* **B**. *(вж. дефиниция 4.3.21).*

Доказателство. φ се удовлетворява от Г тогава и само тогава, когато φ е изпълнима във всеки модел на Г. Съгласно малката теорема на Ербран това се случва тогава и само тогава, когато φ е изпълнима във всеки ербранов модел на Г. Тъй като Г и φ не съдържат символа за равенство, то интерпретацията на този символ е без значение и значи последното се случва тогава и само тогава, когато φ е изпълнима във всеки ербранов модел на Г, в който символа за равенство се интерпретира като равенство. Но една структура е ербранова и символа за равенство в нея се интерпретира като равенство тогава и само тогава, когато тя е обогатяване на **B**. Следователно горното се случва тогава и само тогава, когато φ е изпълнима във всеки модел на Г, който е обогатяване на **B**. Последното е равносилно с това да съществува отговор, с който при програма Г запитването φ се удовлетворява. ∎

4.4.13. Дефиниция. Ако φ е безквантorна формула и s е субституция, заместваща всяка променлива с терм без променливи, то формулата φs се нарича *затворен частен случай* на формулата φ при субституция s.

4.4.14. Теорема („средна“ теорема на Ербран). *Множество* Г *от безквантorни формули е изпълнимо тогава и само тогава, когато е изпълнимо множеството от всички затворени частни случаи на формулите от* Г.

Доказателство. ($\Longrightarrow$) Да допуснем, че формулите от Г са тъждествено верни в структура **M**. Нека φ е произволна формула от Г и

s е субституция. За да докажем, че частният случай φs е тъждествено верен в $\mathbf{M}$, да изберем произволна оценка v в $\mathbf{M}$. Да дефинираме оценката w по следния начин:

$$w(\mathtt{x}) = [\![s(\mathtt{x})]\!]^{\mathbf{M}} v$$

Тъй като формулата φ е тъждествено вярна в $\mathbf{M}$, то $\mathbf{M} \vDash \varphi[w]$. Съгласно лемата за субституциите 3.3.17 последното е еквивалентно на $\mathbf{M} \vDash (\varphi s)[v]$.

($\Longleftarrow$) Да допуснем, че множеството от всички затворени частни случаи на формули от Γ е изпълнимо. Съгласно малката теорема на Ербран 4.4.10, това множество има ербранов модел $\mathbf{H}$. Ще докажем, че този ербранов модел е модел и на Γ. За целта да изберем произволна формула φ от Γ и нека v е произволна оценка в $\mathbf{H}$. Тъй като частният случай φv е верен в $\mathbf{H}$, то от твърдение 4.4.8 получаваме, че формулата φ е вярна в $\mathbf{H}$ при оценка v. ∎

4.5. Алгоритъм за унификация

Унификация в ербранова структура

Да припомним, че в 4.4.11 дефинирахме $\mathbf{B}$ да бъде ербранова структура, в която единственият предикатен символ е $=$ и той се интерпретира по обичайния начин, т.е. като равенство. Теорема 4.4.10 и следствие 4.4.12 показват, че тази структура $\mathbf{B}$ е много интересна от теоретична гледна точка. Тя обаче е интересна и от практическа гледна точка, защото общо взето може да считаме, че точно такава структура $\mathbf{B}$ използва ядрото на езика пролог.

В езика пролог има немалко вградени предикати, за които не може да се даде никаква логическа семантика (такива са например предикатите `!` и `==`). Освен това по-новите версии на пролог предлагат разнообразни предикати за аритметични и други видове ограничения. Ако обаче от някоя по-стара версия на пролог „изхвърлим“ нелогическите предикати, то ще получим език за логическо програмиране с ограничения, използващ ербранова структура $\mathbf{B}$.*

Ще опишем алгоритъм, който може да се използва за решаване на ограничения, когато структурата $\mathbf{B}$ е дефинирана както в 4.4.11. От

*Дори и да изхвърлим „нелогическите“ предикати, в пролог пак ще останат много вградени предикати освен равенството. Това обаче от теоретична гледна точка е без значение, защото има начин тези предикати да се дефинират с логическа програма и значи не е нужно да считаме, че те са вградени.

съображения за ефективност повечето версии на пролог използват леко видоизменен вариант на този алгоритъм, който не винаги работи коректно при ербранова структура $\mathbf{B}$, но не извършва т. н. occurs check и затова е по-ефективен. Вж. края на този раздел.

Да припомним, че съгласно дефиниция 4.3.9 а) ограниченията представляват конюнкции от атомарни формули от сигнатурата ВГРАД. В нашия случай единственият предикатен символ в сигнатурата ВГРАД е $=$, което означава, че ограниченията представляват конюнкции от равенства между термове:

$$\tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \cdots \,\&\, \tau_n = \sigma_n$$

Затова може да си мислим, че ограниченията представляват системи от нула (когато ограничението е $\top$) или повече уравнения, които решаваме в структурата $\mathbf{B}$.

✓ **4.5.1. Дефиниция.** Казваме, че оценката v е *решение на ограничението* φ, ако $\mathbf{B} \vDash \varphi[v]$.

Разбира се, v е решение на дадено ограничение тогава и само тогава, когато всички уравнения в него са верни при оценка v.

4.5.2. Дефиниция. За две ограничения казваме, че са *еквивалентни* в $\mathbf{B}$, ако имат едни и същи решения.

✓ **4.5.3. Дефиниция.** Казваме, че едно ограничение е *решено* относно променливата $\mathtt{x}$, ако тази променлива се среща точно веднъж в ограничението и то в уравнение от вида $\mathtt{x} = \tau$. (От тук в частност следва, че $\mathtt{x}$ не се среща в τ, нито в което и да е друго уравнение от ограничението). За уравнението $\mathtt{x} = \tau$ казваме, че е *решаващо*.

Алгоритъмът за унификация, към чиято дефиниция пристъпваме сега, представлява метод, посредством който за всяко ограничение може да намерим еквивалентно на него ограничение, в което всички уравнения са решаващи.

✓ **4.5.4. Дефиниция.** Следните преобразувания на ограничения ще наричаме *решаващи преобразувания*:

Първо решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\tau = \mathtt{x}$$

където $\mathtt{x}$ е променлива, а термът τ не е променлива, то заменяме това уравнение с

$$\mathtt{x} = \tau$$

Второ решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\mathtt{x} = \mathtt{x}$$

където $\mathtt{x}$ е променлива, то отстраняваме това уравнение от ограничението.

Трето решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

то заменяме това уравнение с

$$\tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \cdots \,\&\, \tau_n = \sigma_n$$

Ако в ограничението има уравнение от вида

$$\mathtt{c} = \mathtt{c}$$

където $\mathtt{c}$ е символ за константа, то отстраняваме това уравнение от ограничението.

Четвърто решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\mathtt{x} = \tau$$

което не е решаващо и променливата $\mathtt{x}$ не се среща в терма τ, то заместваме навсякъде в останалите уравнения променливата $\mathtt{x}$ с τ.

✓ **4.5.5.** Алгоритъмът за унификация се състои в следното: да прилагаме решаващи преобразувания към ограничението по произволен начин докато може. Следващото твърдение показва, че ако към някое ограничение не може повече да прилагаме решаващи преобразувания, то в него със сигурност или всички уравнения ще бъдат решаващи, или ограничението ще съдържа уравнение, което няма да има решения, и значи цялото ограничение няма да има решения.

4.5.6. Твърдение. *Ако към някое ограничение не може да се прилагат решаващи преобразувания, то в него със сигурност или всички уравнения са решаващи, или ограничението съдържа уравнение, което няма решения.*

Доказателство. Ако разгледаме какво представляват решаващите преобразувания, може да забележим, че ако към някое ограничение не може повече да прилагаме такива преобразувания, то със сигурност всяко уравнение в ограничението ще бъде или решаващо, или ще има някой от следните пет вида:

$$\begin{aligned}
&\mathtt{a} = \mathtt{b} && (\mathtt{a} \text{ и } \mathtt{b} \text{ са различни символи})\\
&\mathtt{c} = \mathtt{f}(\dots)\\
&\mathtt{f}(\dots) = \mathtt{c}\\
&\mathtt{f}(\dots) = \mathtt{g}(\dots) && (\mathtt{f} \text{ и } \mathtt{g} \text{ са различни символи})\\
&\mathtt{x} = \tau && (\tau \text{ не е променлива и променливата } \mathtt{x} \text{ се съдържа в } \tau)
\end{aligned}$$

Всяко уравнение от тези пет вида няма решение. За първите четири това е очевидно, защото $\mathbf{B}$ е ербранова структура, а за петото да забележим, че съгласно твърдение 4.4.6 стойността на τ при коя да е оценка v се получава като заменим всяка променлива в τ, включително променливата $\mathtt{x}$, с $v(\mathtt{x})$. Това означава, че броят на символите в стойността на τ при оценка v със сигурност е по-голям, от колкото броят на символите в $v(\mathtt{x})$. ∎

За да има полза от твърдение 4.5.6, трябва да докажем следните две неща: 1) че от каквото и ограничение да започнем и както и да прилагаме решаващите преобразувания, то със сигурност след краен брой стъпки ще получим ограничение, към което повече не може да прилагаме решаващи преобразувания (т.е. алгоритъмът за унификация не се зацикля) и 2) след всяко прилагане на решаващо преобразувание, новото ограничение е еквивалентно в $\mathbf{B}$ на първоначалното (т.е. алгоритъмът за унификация е коректен). Тези две неща са доказани в твърдения 4.5.7 и 4.5.8.

4.5.7. Твърдение. *Не е възможно към едно ограничение да прилагаме решаващи преобразувания до безкрайност.*

Доказателство. Първо да забележим, че ако системата е решена относно някоя променлива, то и след прилагането на някое решаващо преобразувание тя ще остане решена относно тази променлива. От друга страна след прилагането на четвъртото решаващо преобразувание броят на променливите, относно които ограничението е решено, се увеличава с една. Тъй като в ограничението се срещат само краен брой променливи, ясно е, че няма как до безкрайност да увеличаваме броят на променливите, относно които ограничението е решено. Следователно четвъртото решаващо преобразувание може да се прилага само краен

брой пъти. След като приложим четвъртото решаващо преобразуване за последен път получаваме ограничение, към което повече не се прилагат решаващи преобразувания от четвърти вид.

Сега да забележим, че решаващите преобразувания от първи, втори и трети вид не увеличават броя на срещанията на функционални символи и символи за константи в ограничението. От друга страна прилагането на решаващо преобразуване от трети вид със сигурност намалява този брой. Ясно е, че това няма как да става безброй много пъти, следователно третото решаващо преобразуване може да се прилага само краен брой пъти. След като го приложим за последен път, получаваме ограничение, към което повече не се прилагат решаващи преобразувания от трети и четвърти вид.

Самата дефиниция на преобразуванията от първи и втори вид изключва възможността да прилагаме само такъв вид преобразувания безброй пъти. ∎

4.5.8. Твърдение. *След прилагане на решаващо преобразуване получаваме ограничение, което е еквивалентно в* $\mathbf{B}$ *на първоначалното.*

Доказателство. Твърдението е очевидно по отношение на решаващите преобразувания от първи и втори вид.

Ако е било приложено решаващо преобразуване от трети вид за уравнението

$$\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

то също няма нищо сложно — тъй като структурата е ербранова, то това уравнение е вярно точно при онези оценки, при които е вярна конюнкцията

$$\tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \cdots \,\&\, \tau_n = \sigma_n$$

Ако е било приложено решаващо преобразуване от трети вид за уравнението $\mathtt{c} = \mathtt{c}$, то твърдението отново е очевидно.

Нека е било приложено решаващо преобразуване от четвърти вид за уравнението

$$\mathtt{x} = \tau$$

Ако v е решение на ограничението (без значение дали преди или след преобразуването), то $v(\mathtt{x}) = [\![\tau]\!]^{\mathbf{H}} v$. Съгласно лемата за субституциите (твърдение 3.3.13), за произволен терм σ

$$[\![\sigma]\!]^{\mathbf{H}} w = [\![\sigma[\mathtt{x} \coloneqq \tau]]\!]^{\mathbf{H}} v$$

където оценката w е дефинирана с равенството

$$w(\mathtt{z}) = [\![\mathtt{z}[\mathtt{x} \coloneqq \tau]]\!]^{\mathbf{H}} v$$

Но $v(\mathtt{x}) = [\![\tau]\!]^{\mathbf{H}} v$, а когато $\mathtt{z} \neq \mathtt{x}$, то $\mathtt{z}[\mathtt{x} \coloneqq \tau] = \mathtt{z}$ и значи оценките v и w съвпадат. Следователно за произволен терм σ

$$[\![\sigma]\!]^{\mathbf{H}} v = [\![\sigma[\mathtt{x} \coloneqq \tau]]\!]^{\mathbf{H}} v$$

т.е. стойността на кой да е терм при оценка v си остава същата, ако към терма приложим субституцията $\mathtt{x} \coloneqq \tau$. Тъй като четвъртото решаващо преобразуване представлява прилагане на точно тази субституция, то значи v или е решение на ограничението както преди, така и след преобразуването, или v не е решение на ограничението нито преди, нито след преобразуването. ∎

✓ **4.5.9.** В логическото програмиране алгоритъмът за унификация се използва по следния начин. След като го приложим към ограничението на текущото състояние, то или получаваме ограничение, за което знаем, че няма решение, или получаваме ограничение, в което всяко уравнение е решаващо. В първия случай цялото състояние няма решение и повече не го използваме. Във втория случай нека сме получили състояние от вида

$$\langle \texttt{?-}\, \varphi_1, \varphi_2, \ldots, \varphi_k \parallel \mathtt{x}_1 = \tau_1, \mathtt{x}_2 = \tau_2, \ldots, \mathtt{x}_n = \tau_n \rangle$$

Да приложим към всяка от формулите $\varphi_1, \varphi_2, \ldots, \varphi_k$ субституцията

$$\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n$$

Ако означим така получените формули с $\varphi'_1, \varphi'_2, \ldots, \varphi'_k$, то ще получим състоянието

$$\langle \texttt{?-}\, \varphi'_1, \varphi'_2, \ldots, \varphi'_k \parallel \mathtt{x}_1 = \tau_1, \mathtt{x}_2 = \tau_2, \ldots, \mathtt{x}_n = \tau_n \rangle$$

в което никоя от променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ не се среща в запитването $\texttt{?-}\, \varphi'_1, \varphi'_2 \ldots, \varphi'_k$. Това означава, че изпълнението на програмата може да продължи без да се интересуваме повече от ограничението. Чак когато пролог намери отговор на първоначално поставеното запитване от потребителя, може да използваме равенствата в ограничението, за да изведем отговор. И тъй като може да прилагаме алгоритъма за унификация винаги, когато решим, това означава, че изчисленията може да се извършват, използвайки състояния, от чиито ограничения не се интересуваме преди да завърши работата на програмата.

Тук няма да доказваме, че замяната на формулите $\varphi_1, \varphi_2, \ldots, \varphi_k$ с $\varphi'_1, \varphi'_2, \ldots, \varphi'_k$ по описания по-горе начин не променя по същество начина, по който работи една логическа програма, спрямо това, което вече бе описано в раздел 4.3. Ще се задоволим да докажем семантичната коректност на тази замяна. Тази коректност следва от следното твърдение:

4.5.10. Твърдение. *Нека променливите* $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ *не се срещат в термовете* $\tau_1, \tau_2, \ldots, \tau_n$. *Ако структурата* $\mathbf{M}$ *е обогатяване на* $\mathbf{B}$, *то формулата*

$$\varphi \,\&\, \mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \cdots \,\&\, \mathtt{x}_n = \tau_n$$

е вярна в $\mathbf{M}$ *при някоя оценка* v *тогава и само тогава, когато при същата оценка е вярна формулата*

$$\varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n] \,\&\, \mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \cdots \,\&\, \mathtt{x}_n = \tau_n$$

Доказателство. Доказателството е аналогично на разглеждането на решаващите преобразувания от четвърти тип в твърдение 4.5.8, само че прилагаме лемата за субституциите за формули, а не за термове.

Ако някоя от двете формули в условието на твърдението е вярна при оценка v, то $v(\mathtt{x}_i) = [\![\tau_i]\!]^{\mathbf{H}} v$. Съгласно лемата за субституциите (твърдение 3.3.17), за формулата φ е вярно

$$\mathbf{M} \vDash \varphi[w] \longleftrightarrow \mathbf{M} \vDash \varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n][v]$$

където оценката w е дефинирана с равенството

$$w(\mathtt{z}) = [\![\mathtt{z}[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n]]\!]^{\mathbf{H}} v$$

Но $v(\mathtt{x}_i) = [\![\tau_i]\!]^{\mathbf{H}} v$, а когато $\mathtt{z} \notin \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\}$, то

$$\mathtt{z}[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n] = \mathtt{z}$$

и значи оценките v и w съвпадат. Следователно за формулата φ е имаме

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{M} \vDash \varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n][v]$$

От тук следва исканото. ∎

Примери

Да разгледаме с конкретен пример как работи пролог.

✓ **4.5.11. Пример.** Нека програмата се състои от следните клаузи:

$$\mathtt{p(a)} \mathtt{:-} \mathtt{p(c)}. \tag{54}$$

$$\mathtt{p}(X) \mathtt{:-} \mathtt{r}(X). \tag{55}$$

$$\mathtt{r(a)}. \tag{56}$$

Ще приложим тази програма към запитването

$$\texttt{?- p(a)}$$

Клауза (54) свежда първоначалното състояние

$$\langle \texttt{?- p(a)} \parallel \top \rangle \tag{57}$$

към състоянието

$$\langle \texttt{?- p(c)} \parallel \texttt{a} = \texttt{a} \rangle$$

Като приложим третото решаващо преобразуване към $\texttt{a} = \texttt{a}$, това състояние се опростява до

$$\langle \texttt{?- p(c)} \parallel \top \rangle$$

Единствената клауза, която не свежда това състояние към състояние с неизпълнимо ограничение, е клауза (55). Посредством нея то се свежда до

$$\langle \texttt{?- r}(X_1) \parallel \texttt{c} = X_1 \rangle$$

Алгоритъмът за унификация свежда това ограничение към $X_1 = \texttt{c}$. След прилагане на субституцията $X_1 := \texttt{c}$ към запитването, променливата X_1 става излишна, така че състоянието се опростява до

$$\langle \texttt{?- r(c)} \parallel \top \rangle$$

Никоя клауза не може да се използва за това запитване, така че то остава неудовлетворено.

За запитването (57) имаме още една възможност — да приложим клауза (55). Посредством нея то се свежда до

$$\langle \texttt{?- r}(X_2) \parallel \texttt{a} = X_2 \rangle$$

Алгоритъмът за унификация свежда това ограничение към $X_2 = \texttt{a}$. След прилагане на субституцията $X_1 := \texttt{a}$ към запитването, променливата X_2 става излишна, така че състоянието се опростява до

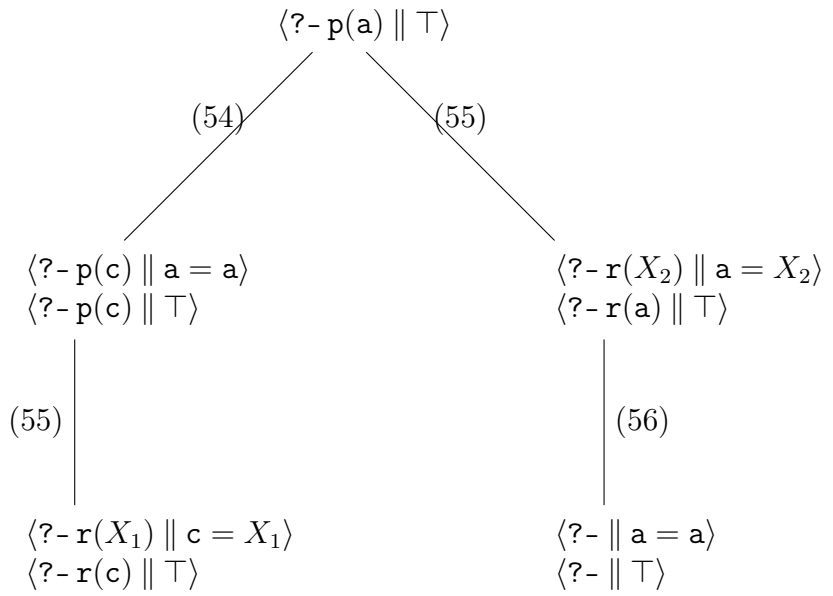
$$\langle \texttt{?- r(a)} \parallel \top \rangle$$

Клауза (56) свежда това състояние до

$$\langle \texttt{?-} \parallel \texttt{a} = \texttt{a} \rangle$$

Третото решаващо преобразуване елиминира равенството $\texttt{a} = \texttt{a}$, така че получаваме успешното състояние

$$\langle \texttt{?-} \parallel \top \rangle$$



Фиг. 14.

Всички свеждания от пример 4.5.11 могат да бъдат изобразени с дърво, както това е направено във фигура 14. Във върховете на това дърво сме записали по две състояния. Първото се получава преди да приложим алгоритъмът за унификация, а второто — след това. Пълнотата означава, че ако първоначалната заявка се удовлетворява, то това дърво със сигурност ще съдържа успешен клон (т.е. клон, чиито състояния представляват извод).

Интерпретаторът на пролог обхожда това дърво в дълбочина, започвайки от най-левите клонове. В случая пролог ще обходи най-напред левия клон на дървото, където няма да намери решение на задачата, и след това ще се прехвърли на десния клон.

Когато в дървото има безкраен ляв клон, пролог ще се зацикли без да открие успешен клон. Да разгледаме още един пример.

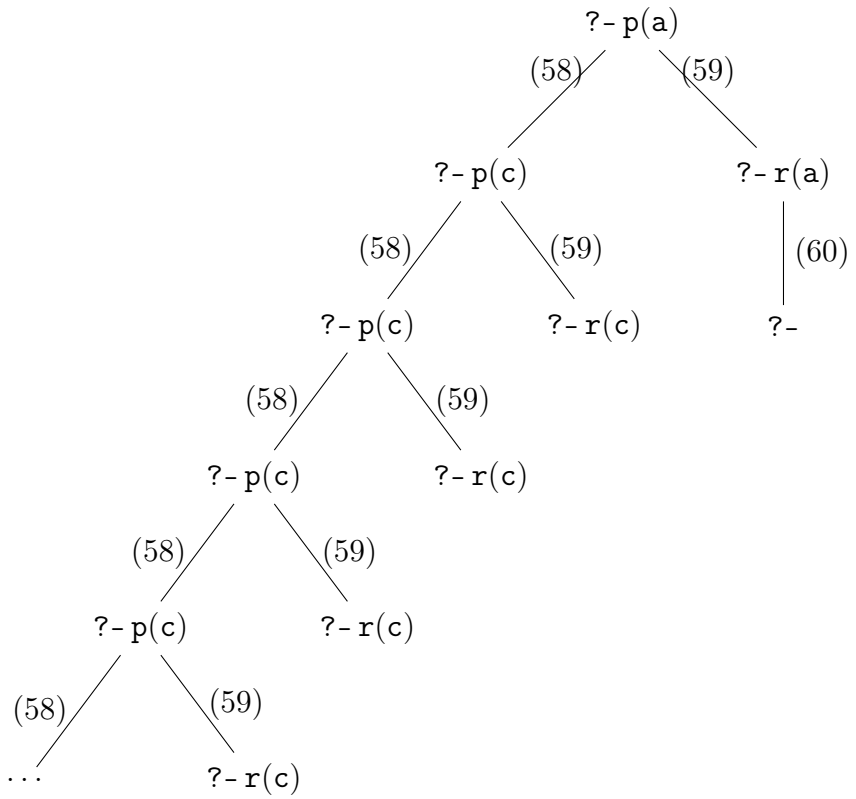
✓ **4.5.12. Пример.** Нека програмата този път се състои от следните клаузи:

$$\mathtt{p}(X)\ \mathtt{:-}\ \mathtt{p}(\mathtt{c}). \tag{58}$$

$$\mathtt{p}(X)\ \mathtt{:-}\ \mathtt{r}(X). \tag{59}$$

$$\mathtt{r}(\mathtt{a}). \tag{60}$$

Дървото на извод, което се получава от тази програма при запитване ?- p(a), е изобразено на фигура 15. За по-добра прегледност, в това дър-

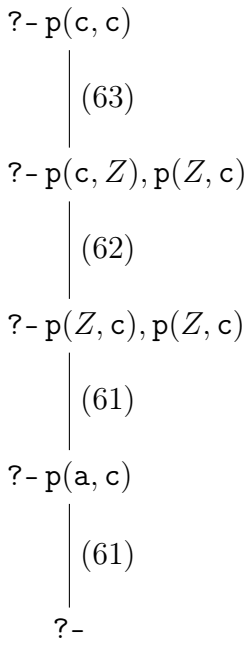


Фиг. 15.

во са изобразени само запитванията в състоянията, които се получават след прилагане на алгоритъма за унификация. Вижда се, че това дърво съдържа безкраен ляв клон. Пролог ще тръгне по този клон и никога няма да открие десния и успешен клон.

Това показва, че при пролог е от значение редът, в който са записани клаузите. Когато дървото на извод е крайно, тогава при лошо подреждане на клаузите пролог може да работи значително по-бавно. Например при дървото на фигура 14 пролог ненужно хаби време, за да обхожда левия и неуспешен клон. Много по-лошо е положението, когато дървото е безкрайно. В този случай пролог може до безкрайност да се движи по някой безкраен клон, както се случи при дървото от фигура 15.

В първата от тези програми проблемът се оправя, ако се разменят клаузи (54) и (55), а във втората — като се разменят клаузи (58) и (59). За съжаление обаче, в някои случаи пролог се зацикля без значение в какъв ред подреждаме клаузите.



Фиг. 16.

Да разгледаме още един пример.*

✓ **4.5.13. Пример.** Нека програмата се състои от следните клаузи:

$$\texttt{p(a, c)}. \tag{61}$$
$$\texttt{p}(X, Y)\ \texttt{:-}\ \texttt{p}(Y, X). \tag{62}$$
$$\texttt{p}(X, Y)\ \texttt{:-}\ \texttt{p}(X, Z),\ \texttt{p}(Z, Y). \tag{63}$$

По принцип запитването `?- p(c, c)` може да се удовлетвори с обратен извод. Едно успешно свеждане на това запитване до празното запитване е дадено на фигура 16. Въпреки това, както и да подреждаме клаузите в тази програма, винаги ще се оказва, че отляво на този успешен клон, а и изобщо отляво на кой да е успешен клон, в дървото ще има безкраен клон. Затова при тази програма пролог ще се зацикля без значение как подреждаме клаузите в програмата.

Унификация без occurs check

Ако разгледаме решаващите преобразувания (вж. дефиниция 4.5.4) от алгоритъма за унификация, може да забележим, че първото, второто

*Автор на примера е проф. Димитър Скордев [26].

и третото решаващо преобразование могат да се изпълнят на компютър за константно време (т.е. време, което не зависи от размера на термовете). При четвъртото решаващо преобразование имаме уравнение от вида $\mathtt{x} = \tau$ и трябва да проверим дали променливата $\mathtt{x}$ се среща в терма τ. Тази проверка, наречена на английски occurs check, не може да се извърши за константно време, защото зависи от големината на терма τ. Когато термът τ представлява някаква голяма структура данни, тази проверка може да отнеме доста време, а това е нежелателно, тъй като скоростта на работа на пролог зависи до голяма степен от това колко бързо компютърът може да унифицира термове.

Най-простият начин да подобрим ефективността на алгоритъма за унификация е просто да не проверяваме дали променливата $\mathtt{x}$ се среща в τ. По този начин всички решаващи преобразувания ще могат да се изпълняват за константно време, но за съжаление в някои случаи алгоритъмът ще зацикля. Да разгледаме пример когато това се случва.

4.5.14. Пример. Да разгледаме следната система:

$$\mathtt{x} = \mathtt{f}(\mathtt{x}), \mathtt{y} = \mathtt{f}(\mathtt{y}), \mathtt{x} = \mathtt{y}$$

Не е трудно да се забележи, че ако приложим алгоритъмът за унификация към тази система без да извършваме occurs check, то алгоритъмът за унификация може да зацикли например по следния начин:

$$\begin{gathered}
\mathtt{x} = \mathtt{f}(\mathtt{x}), \mathtt{y} = \mathtt{f}(\mathtt{y}), \mathtt{x} = \mathtt{y} \\
\mathtt{y} = \mathtt{f}(\mathtt{y}), \mathtt{f}(\mathtt{x}) = \mathtt{y}, \mathtt{x} = \mathtt{f}(\mathtt{x}) \\
\mathtt{f}(\mathtt{x}) = \mathtt{f}(\mathtt{y}), \mathtt{x} = \mathtt{f}(\mathtt{x}), \mathtt{y} = \mathtt{f}(\mathtt{y}) \\
\mathtt{x} = \mathtt{f}(\mathtt{x}), \mathtt{y} = \mathtt{f}(\mathtt{y}), \mathtt{x} = \mathtt{y} \\
\mathtt{y} = \mathtt{f}(\mathtt{y}), \mathtt{f}(\mathtt{x}) = \mathtt{y}, \mathtt{x} = \mathtt{f}(\mathtt{x}) \\
\mathtt{f}(\mathtt{x}) = \mathtt{f}(\mathtt{y}), \mathtt{x} = \mathtt{f}(\mathtt{x}), \mathtt{y} = \mathtt{f}(\mathtt{y}) \\
\mathtt{x} = \mathtt{f}(\mathtt{x}), \mathtt{y} = \mathtt{f}(\mathtt{y}), \mathtt{x} = \mathtt{y} \\
\ldots\ldots\ldots
\end{gathered}$$

За да не се случва такова зацикляне, алгоритъмът за унификация на пролог използва по-различни рашаващи преобразувания.

4.5.15. Дефиниция (решаващи преобразувания на пролог). Следните преобразувания на ограничения ще наричаме *решаващи преобразувания на пролог*:

Първо решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\tau = \mathtt{x}$$

където $\mathtt{x}$ е променлива, а термът τ не е променлива, то заменяме това уравнение с

$$\mathtt{x} = \tau$$

Второ решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\mathtt{x} = \mathtt{x}$$

където $\mathtt{x}$ е променлива, то отстраняваме това уравнение от ограничението.

Трето решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

то заменяме това уравнение с

$$\tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \cdots \,\&\, \tau_n = \sigma_n$$

Ако в ограничението има уравнение от вида

$$\mathtt{c} = \mathtt{c}$$

където $\mathtt{c}$ е символ за константа, то отстраняваме това уравнение от ограничението.

Четвърто решаващо преобразуване. Ако в ограничението има две уравнения от вида

$$\mathtt{x} = \tau, \mathtt{x} = \sigma$$

то заместваме тези уравнения с

$$\mathtt{x} = \tau, \tau = \sigma$$

Всеки терм си има синтактично дърво. Затова стандартният алгоритъм за унификация може да се използва не само за унификация на термове, но и за унификация на дървета. Въпреки че когато унифицираме термове или дървета горните решаващи преобразувания не работят коректно, оказва се че те работят правилно, ако универсумът на структурата съдържа не само дървета, ами произволни графи.* Затова когато програмистът работи умело, той може да счита, че графите са вграден тип данни впролог.

*Става въпрос за графи, чиито върхове са етикетирани с функционални символи, и от всеки връх етикетиран с n-местен функционален символ излизат n ребра с определена подредба (т.е. ясно е кое е първото ребро, кое второто и т.н.).

4.6. Метод на резолюциите

Дизюнкти

В предходните раздели разработихме различни начини, посредством които компютрите могат да „разсъждават“. Общото на тези начини бе това, че те използват клаузи и цели. Затова съвсем естествено е да се запитаме: можем ли да преведем на езика на клаузите и целите всеки въпрос, формулиран посредством предикатната логика?

Да си припомним следната дефиниция:

4.6.1. Дефиниция. а) *Литерал* означава атомарна формула или отрицание на атомарна формула.

б) *Елементарна дизюнкция* означава безквантворна формула, в която единствените логически операции са дизюнкция и отрицание и всяко отрицание се намира пред атомарна формула.

Да си припомним твърдение 3.7.31. То казва, че когато се интересуваме от изпълнимостта на множество от произволни формули и използваме класическата логика, може да считаме, че елементите на това множество имат сравнително прост вид — елементарни дизюнкции. Следователно ако успеем да покажем, че с помощта на клаузи и цели сме в състояние да изразим прости формули като елементарните дизюнкции, оттук би следвало, че и въпроси, формулирани посредством произволни формули, могат да бъдат сведени към клаузи и цели.

За съжаление, оказва се, че това не е възможно. С помощта на клаузи и цели могат да се представят не произволни, а само т. н. хорнови елементарни дизюнкции.

Да видим защо това е така.

4.6.2. Дефиниция. Една елементарна дизюнкция е *хорнова*, ако в нея има най-много един литерал без отрицание.

✓ **4.6.3. Твърдение.** *Нека* Γ *е множество от клаузи и* ζ *е цел. Може да се намери такова множество от хорнови елементарни дизюнкции,*[*] *че при използване на некласическата логика то е неизпълнимо тогава и само тогава, когато целта* ζ *се удовлетворява при програма* Γ*.*

✓ Доказателство. Нека Γ е множеството от клаузи и ζ е цел, за която се питаме дали се удовлетворява при програма Γ. По дефиниция

[*]Смисълът тук е това, че това множество от елементарни дизюнкции може да бъде намерено конструктивно, т. е. посредством алгоритъм.

да се удовлетворява ζ означава за всеки модел на Γ да има оценка v, при която целта ζ е вярна (дефиниция 4.4.1). Ако разсъждаваме неконструктивно, това е така тогава и само тогава, когато не съществува модел на Γ, при който ζ е лъжа при всяка оценка. С други думи, тогава и само тогава, когато множеството $\Gamma \cup \{\neg\zeta\}$ е неизпълнимо.

В тривиалния случай, когато ζ е празната цел, то по дефиниция 4.1.3, $\zeta = \top$, значи $\neg\zeta$ е винаги невярна формула, следователно множеството $\Gamma \cup \{\neg\zeta\}$ е неизпълнимо все едно каква е програмата Γ. Значи ще ни свърши работа кое да е неизпълнимо множество от хорнови елементарни дизюнкции.*

Остава да разгледаме случая когато ζ е непразна цел. Да видим как ще изглеждат елементите на множеството $\Gamma \cup \{\neg\zeta\}$ когато ги изразим посредством елементарни дизюнкции.

Елементите на Γ са клаузи. Всяка клауза от вида

$$\psi \texttt{ :- } \varphi_1, \varphi_2, \ldots, \varphi_n$$

при $n \geqq 1$ представлява различен запис на формулата

$$\varphi_1 \,\&\, \varphi_2 \,\&\, \cdots \,\&\, \varphi_n \Rightarrow \psi$$

Ако преобразуваме тази формула в конюнктивна нормална форма по начина, описан в доказателствата на твърдения 3.7.8 и 3.7.30, ще получим формула, състояща се от една единствена елементарна дизюнкция:

$$\neg\varphi_1 \vee \neg\varphi_2 \vee \cdots \vee \neg\varphi_n \vee \psi$$

Когато пък $n = 0$, то клаузата представлява просто формулата ψ и значи пак е елементарна дизюнкция от горния вид (само че $n = 0$). Специфичното при така получената елементарна дизюнкция е това, че тя съдържа произволен брой (може и нула) литерали с отрицание, но точно един литерал без отрицание. И така, направихме следното важно наблюдение: всяка клауза може да бъде представена посредством хорнова елементарна дизюнкция с точно един положителен литерал

Да видим сега как можем да представим $\neg\zeta$ посредством елементарни дизюнкции. Тъй като вече разгледахме случая, когато ζ е празната цел, то ζ има вида

$$\texttt{?- } \varphi_1, \varphi_2, \ldots, \varphi_n \quad (n \geqq 0)$$

*Например ако в сигнатурата има едноместен предикатен символ `p` и символ за константа `c`, то множеството $\{\texttt{p(c)}, \neg\texttt{p(c)}\}$ е неизпълнимо множество от две елементарни дизюнкции.

По дефиниция тази цел е различен запис на формулата

$$\varphi_1 \,\&\, \varphi_2 \,\&\, \cdots \,\&\, \varphi_n$$

Като сложим отпред отрицание, получаваме

$$\neg(\varphi_1 \,\&\, \varphi_2 \,\&\, \cdots \,\&\, \varphi_n)$$

След като преобразуваме тази формула в конюнктивна нормална форма по начина, описан в доказателствата на твърдения 3.7.8 и 3.7.30, получаваме елементарната дизюнкция

$$\neg\varphi_1 \vee \neg\varphi_2 \vee \cdots \vee \neg\varphi_n$$

Специфичното при тази елементарна дизюнкция е това, че тя съдържа произволен брой литерали с отрицание и не съдържа нито един литерал без отрицание. Следователно тази елементарна дизюнкция също е хорнова. ■

И така, направихме следното важно наблюдение: при използване на класическата логика, въпросът, дали дадена цел се удовлетворява при някакво множество от клаузи, е еквивалентен на въпроса, дали множество от хорнови елементарни дизюнкции е неизпълнимо.

В този раздел ще опишем методът на резолюциите, посредством който компютрите могат да „разсъждават“, използвайки произволни елементарни дизюнкции, а не само хорнови. Но за да ни бъде по-удобно, вместо с елементарни дизюнкции, ще работим с по-просто устроени обекти, наречени дизюнкти.

✓ **4.6.4. Дефиниция.** а) *Дизюнкт* е крайно (може и празно) множество от литерали.*

б) Един дизюнкт е неверен в структурата $\mathbf{M}$ при оценка v, ако всичките му литерали са неверни в $\mathbf{M}$ при оценка v.

в) Един дизюнкт е верен в структурата $\mathbf{M}$ при оценка v, ако не е вярно, че е неверен в $\mathbf{M}$ при оценка v.

г) Един дизюнкт е (тъждествено) верен в структура $\mathbf{M}$, ако е верен при всяка оценка в $\mathbf{M}$.

*Терминът „дизюнкт“ има такъв смисъл само в България, докато в руската математическа литература „дизъюнкт“ означава елементарна дизюнкция. Това, което тук сме дефинирали като „дизюнкт“, извън България се нарича „клауза“ (clause). Разбира се, тъй като нещата, които тук наричаме „клаузи“, в другите страни също се наричат клаузи, то от контекста трябва да познаваме за какъв вид клаузи става въпрос.

д) Множество от дизюнкти е *изпълнимо*, ако съществува структура $\mathbf{M}$, в която са тъждествено верни всички дизюнкти от множеството. Структурата $\mathbf{M}$ е *модел* на това множество.

е) Множество от дизюнкти е *неизпълнимо*, ако не е вярно, че то е изпълнимо (т.е. ако няма модел).

4.6.5. *Забележка*: Току-що дадената дефиниция показва, че дизюнкт от вида

$$\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$$

е еквивалентен на формулата

$$\neg\neg(\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_n)$$

Разбира се, когато разсъждаваме, използвайки класическата логика, тази формула е еквивалентна на елементарната дизюнкция

$$\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_n$$

Тъй като дизюнктът $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ по отношение на класическата логика е еквивалентен на елементарната дизюнкция $\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_n$, то значи от твърдение 3.7.31 моментално получаваме следното следствие:

4.6.6. Следствие. *Ако в сигнатурата разполагаме с достатъчно неизползвани символи за константи и функционални символи, то за всяко крайно множество*[*] *от формули можем да намерим такова крайно множество от дизюнкти, че (от гледна точка на класическата логика) първоначалното множество е изпълнимо тогава и само тогава, когато е изпълнимо множеството от дизюнкти.*

По подобен начин може да установим, че много от нещата, които сме доказали за формули, са верни и когато използваме дизюнкти. Например от „малката“ теорема на Ербран за безквантворни формули може да получим като следствие „малка“ теорема на Ербран за дизюнкти:

4.6.7. Теорема („малка“ теорема на Ербран за дизюнкти). *Множество* Γ *от дизюнкти има модел тогава и само тогава, когато то има ербранов модел.*

Доказателство. Празният дизюнкт е еквивалентен на формулата $\bot$. Освен това всеки дизюнкт

$$\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$$

[*]Твърдението е вярно и за безкрайни множества, но тук няма да доказваме това.

е еквивалентен на формулата*

$$\neg\neg(\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_n)$$

Горните две наблюдения показват, че можем лесно да изведем тази теорема като следствие от „малката“ теорема на Ербран за безквантор-ни формули 4.4.10. Нека Γ' е множеството от формулите, съответстващи на дизюнктите от Γ. Множеството Γ е изпълнимо тогава и само тогава, когато е изпълнимо Γ', защото дизюнктите от Γ са еквивалентни на съответните формули от Γ'. Съгласно „малката“ теорема на Ербран за безквантор-ни формули, множеството Γ' има модел тогава и само тога-ва, когато то има ербранов модел. Следователно също и множеството Γ има модел тогава и само тогава, когато то има ербранов модел. ■

Следващото твърдение е упражнение как да се прави отрицание.

✓ **4.6.8. Твърдение.** *Един дизюнкт е тъждествено верен в структу-ра* $\mathbf{M}$ *тогава и само тогава, когато не съществува оценка в* $\mathbf{M}$*, при която всички литерали от дизюнкта са неверни.*

✓ Доказателство. Да допуснем, че дизюнктът δ е тъждествено ве-рен в $\mathbf{M}$. Трябва да докажем, че не съществува оценка в $\mathbf{M}$, при която всички литерали от дизюнкта са неверни. Ами да допуснем, че такава оценка съществува, т. е. оценка, при която всички литерали са неверни. Съгласно дефиниция 4.6.4 това означава, че δ е неверен в $\mathbf{M}$ при тази оценка, значи δ не е тъждествено верен в $\mathbf{M}$.

Обратната посока също е бърза. Да допуснем, че не съществува оценка в $\mathbf{M}$, при която всички литерали от дизюнкта са неверни. Съг-ласно дефиниция 4.6.4 това означава, че не съществува оценка в $\mathbf{M}$, при която дизюнктът δ е неверен. Значи за всяка оценка v не е вярно, че дизюнктът δ е неверен в $\mathbf{M}$ при тази оценка. Следователно δ е верен в $\mathbf{M}$ при всяка оценка. ■

4.6.9. Дефиниция. Празното множество от литерали се нарича *праз-ен дизюнкт*. За да бъде по-ясно, че работим не с какво да е празно множество, а с дизюнкт, вместо с $\varnothing$, означаваме празния дизюнкт с някой от символите $\square$, ■ или $\{\}$.

Следващото твърдение показва, че празният дизюнкт има същия смисъл, както винаги невярната формула $\bot$.

*Разбира се, когато използваме класическата логика, можем да премахнем двой-ното отрицание.

4.6.10. Твърдение. *Празният дизюнкт не е тъждествено верен в никоя структура.*

Доказателство. Ами да си изберем структура $\mathbf{M}$ и оценка v. Вярно ли е, че в дизюнкта можем да намерим литерал, който е верен при така избраните оценка и структура? Ами очевидно не, защото в празния дизюнкт не можем да намерим какъвто и да е литерал. Щом празният дизюнкт не съдържа верен литерал, значи всичките му литерали са неверни, което по дефиниция означава, че самият празен дизюнкт е неверен. ∎

Либерална резолюция

Либералната резолюция е може би най-простият резолютивен метод, който работи с произволни дизюнкти и за който може да се докажат теореми за коректност и пълнота. Затова в много курсове по логика това е и единственият резолютивен метод, който се разглежда.

Да припомним, че в дефиниция 4.2.10 дефинирахме какво значи частен случай на клауза при оценка. Нека например оценката v е такава, че $v(\mathtt{x}) = 5$ и $v(\mathtt{y}) = 8$. Тогава частният случай при тази оценка на клаузата

$$\mathtt{p}(\mathtt{x}, \mathtt{y}) \mathtt{:-} \mathtt{q}(\mathtt{x})$$

е псевдоклаузата

$$\mathtt{p}(5, 8) \mathtt{:-} \mathtt{q}(5)$$

Няма никакъв проблем да дефинираме частните случаи за дизюнктите по аналогичен начин. Също както при изводимостите с клаузи имахме структура $\mathbf{B}$ с произволен универсум и вградени функционални и предикатни символи, така и при метода на резолюциите може да имаме такава структура и да докажем теоремите за коректност и пълнота по отношение на структура $\mathbf{B}$, даваща семантика на „вградените“ функционални символи. При метода на резолюциите обаче приложенията на неербранови структури $\mathbf{B}$ не са много. Когато структурата $\mathbf{B}$ е ербранова, частните случаи на една клауза се получават като заменим променливите в нея елементи на носителя на $\mathbf{B}$, т.е. с термове без променливи. За по-просто обаче, при либералната резолюция частните случаи се дефинират като заменяме променливите с произволни термове.*

*Въпреки това, ако в следващата дефиниция поискаме термовете да бъдат без променливи, то методът ще остане коректен и пълен.

4.6.11. Дефиниция. а) Нека s е субституция, а ε — дизюнкт. Дизюнктът, който се получава като заменим всеки литерал λ от ε с λs, се нарича *резултат от прилагането* на субституцията s към ε. Ще го означаваме с εs.

б) Всеки дизюнкт от вида εs, където s е произволна субституция, а ε — дизюнкт, се нарича *частен случай* на дизюнкта ε при субституцията s.

4.6.12. Пример. Частният случай на дизюнкта

$$\{\mathtt{p}(\mathtt{x}, \mathtt{f}(\mathtt{c})), \neg\mathtt{q}(\mathtt{f}(\mathtt{y}), \mathtt{x})\}$$

при субституцията $[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{f}(\mathtt{c}), \mathtt{a}]$ е дизюнктът

$$\{\mathtt{p}(\mathtt{f}(\mathtt{c}), \mathtt{f}(\mathtt{c})), \neg\mathtt{q}(\mathtt{f}(\mathtt{a}), \mathtt{f}(\mathtt{c}))\}$$

✓ ***Задача* 50:** Вярно ли е, че всеки частен случай на дизюнкт ε съдържа точно толкова литерали, колкото и ε?

Съгласно „малката“ теорема на Ербран 4.6.7, едно множество от дизюнкти има модел тогава и само тогава, когато то има ербранов модел. А следващото твърдение показва, че когато работим в ербранова структура, всеки дизюнкт е еквивалентен на множеството от онези негови частни случаи, които не съдържат променливи.

4.6.13. Твърдение. *Дизюнктът ε е верен в ербрановата структура* $\mathbf{H}$ *при оценка v тогава и само тогава, когато εv (т. е. частният случай на ε при оценка v) е верен в* $\mathbf{H}$.

Доказателство. Дизюнктът ε е верен в $\mathbf{H}$ при оценка v тогава и само тогава, когато не е невярно, че в ε има литерал, който е верен в $\mathbf{H}$ при оценка v.

Частният случай εv е верен в $\mathbf{H}$ при някаква оценка w тогава и само тогава, когато не е невярно, че в εv има литерал, който е верен в $\mathbf{H}$ при оценка w.

Литералите на εv са точно литералите от вида λv, където $\lambda \in \varepsilon$. Съгласно твърдение 4.4.8, литералът λ е верен в $\mathbf{H}$ при оценка v тогава и само тогава, когато λv е верен в $\mathbf{H}$ (все едно при каква оценка, защото няма променливи). ∎

Идеята на метода на резолюциите е следната. Най-напред използваме следствие 4.6.6, за да сведем първоначалния въпрос, от който се интересуваме, към въпроса, дали някакво множество от дизюнкти е

неизпълнимо. След това започваме да генерираме всевъзможни резолвенти. Ако след краен брой стъпки се стигне до празния дизюнкт, то значи множеството от дизюнкти е неизпълнимо. В противен случай то е изпълнимо.

✓ **4.6.14. Дефиниция.** а) Нека са дадени дизюнктите

$$\{\varphi, \lambda'_1, \lambda'_2, \ldots, \lambda'_n\} \text{ и } \{\neg\varphi, \lambda''_1, \lambda''_2, \ldots, \lambda''_k\}$$

За дизюнкта $\{\lambda'_1, \lambda'_2, \ldots, \lambda'_n, \lambda''_1, \lambda''_2, \ldots, \lambda''_k\}$ ще казваме, че е тяхна *непосредствена резолвента*.

б) *Либерална резолвента* или просто *резолвента* на два дизюнкта ще рече непосредствена резолвента на кои да е частни случаи на тези дизюнкти.

Разбира се, нищо няма да се промени, ако преформулираме дефиницията на непосредствена резолвента по следния начин:

Дефиниция. Нека са дадени дизюнктите

$$\{\varphi\} \cup \delta' \text{ и } \{\neg\varphi\} \cup \delta''$$

За дизюнкта $\delta' \cup \delta''$ ще казваме, че е тяхна *непосредствена резолвента*.

Интуитивно, резолвентата се прави, като „задраскаме“ една двойка противоположни литерали от двата дизюнкта и обединим останалите.

✓ **4.6.15. Пример.** Нека

$$\delta_1 = \{\mathtt{p(c)}, \mathtt{q(c)}, \mathtt{r(a)}\}$$

и

$$\delta_2 = \{\neg\mathtt{p(c)}, \neg\mathtt{q(c)}, \mathtt{r(b)}\}$$

Ако „задраскаме“ двойката литерали $\mathtt{p(c)}$ и $\neg\mathtt{p(c)}$ получаваме непосредствената резолвента

$$\{\mathtt{q(c)}, \mathtt{r(a)}, \neg\mathtt{q(c)}, \mathtt{r(b)}\}$$

Ако пък „задраскаме“ двойката $\mathtt{q(c)}$ и $\neg\mathtt{q(c)}$ получаваме непосредствената резолвента

$$\{\mathtt{p(c)}, \mathtt{r(a)}, \neg\mathtt{p(c)}, \mathtt{r(b)}\}$$

Важно е да се разбере, че дефиницията не ни позволява да „задраскаме“ едновременно и двете двойки $\mathtt{p(c)}$ и $\neg\mathtt{p(c)}$ и $\mathtt{q(c)}$ и $\neg\mathtt{q(c)}$. Затова дизюнктът $\{\mathtt{r(a)}, \mathtt{r(b)}\}$ не е резолвента.

✓ **4.6.16. Пример.** Да разгледаме дизюнктите

$$\varepsilon' = \{\mathtt{p}(\mathtt{x}), \mathtt{p}(\mathtt{f}(\mathtt{c})), \mathtt{q}(\mathtt{x})\}$$
$$\varepsilon'' = \{\neg\mathtt{p}(\mathtt{f}(\mathtt{y}))\}$$

Като приложим субституцията $[\mathtt{x} := \mathtt{f}(\mathtt{y})]$ към ε', виждаме, че

$$\delta_1 = \{\mathtt{p}(\mathtt{f}(\mathtt{y})), \mathtt{p}(\mathtt{f}(\mathtt{c})), \mathtt{q}(\mathtt{f}(\mathtt{y}))\}$$

е частен случай на ε'. Като приложим субституцията–идентитет към ε'', виждаме, че ε'' е частен случай на ε''. Дизюнктите δ_1 и ε'' имат непосредствена резолвента

$$\{\mathtt{p}(\mathtt{f}(\mathtt{c})), \mathtt{q}(\mathtt{f}(\mathtt{y}))\}$$

която е либерална резолвента на ε' и ε''.

Дизюнктът

$$\delta_2 = \{\mathtt{p}(\mathtt{f}(\mathtt{c})), \mathtt{q}(\mathtt{f}(\mathtt{c}))\}$$

също е частен случай на ε', който се получава като приложим към ε' субституцията $[\mathtt{x} := \mathtt{f}(\mathtt{c})]$. Дизюнктите δ_2 и ε'' имат непосредствена резолвента

$$\{\mathtt{q}(\mathtt{f}(\mathtt{c}))\}$$

която е друга либерална резолвента на ε' и ε''.

✓ **4.6.17. Дефиниция.** а) Нека Γ е множество от дизюнкти. *Резолютивен извод* на дизюнкта δ от Γ ще наричаме редица от дизюнкти

$$\delta_0, \delta_1, \delta_2, \ldots, \delta_n$$

в която $\delta_n = \delta$ и за всеки от дизюнктите в редицата е вярно поне едно от следните две условия: 1) че е елемент на Γ или 2) че е резолвента на дизюнкти, които се срещат по-рано в редицата.

б) Дизюнктът δ *се извежда* от Γ с резолюция, ако съществува резолютивен извод на δ от Γ. Ще означаваме това така: $\Gamma \vdash \delta$.

Пристъпваме към доказателство на теореми за коректност и пълнота на либералната резолюция. Започваме с теоремата за коректност. Тя гласи, че ако едно множество от дизюнкти е изпълнимо, то е невъзможно от него да изведем празния дизюнкт.

Както обикновено, доказателството на теорема за коректност ще има следната структура: допускаме, че дизюнктите, с които започваме, са тъждествено верни в някоя структура, и след това показваме, че всички резолвенти, които може да получим, също са верни в тази структура. От това ще следва, че е невъзможно да получим празния дизюнкт, защото той не е верен в никоя структура (вж. твърдение 4.6.10).

✓ **4.6.18. Лема.** *Непосредствената резолвента на дизюнкти, които са верни в структура* $\mathbf{M}$*, също е вярна в* $\mathbf{M}$*.*

✓ Доказателство. Нека дизюнктите $\{\varphi\} \cup \delta'$ и $\{\neg\varphi\} \cup \delta''$ са верни в $\mathbf{M}$. За да докажем, че резолвентата $\delta' \cup \delta''$ е вярна в $\mathbf{M}$, да изберем произволна оценка v в $\mathbf{M}$ и да допуснем, че резолвентата $\delta' \cup \delta''$ не е вярна в $\mathbf{M}$ при оценка v. Съгласно дефиниция 4.6.4 това означава, че при тази оценка всички литерали в $\delta' \cup \delta''$ са неверни. Следователно нито δ', нито δ'' съдържа верен литерал. Но по условие дизюнктът $\{\varphi\} \cup \delta'$ е верен в $\mathbf{M}$ при оценка v, следователно не е вярно, че всичките му литерали са неверни, и значи не е вярно, че литералът φ е неверен. Аналогично и дизюнктът $\{\neg\varphi\} \cup \delta''$ е верен в $\mathbf{M}$ при оценка v, следователно не е вярно, че всичките му литерали са неверни, и значи не е вярно, че литералът $\neg\varphi$ е неверен. Доказахме, че никой от литералите φ и $\neg\varphi$ не е неверен в $\mathbf{M}$ при оценка v, а това е противоречие. ∎

✓ **4.6.19. Лема.** *Частните случаи на дизюнкт, който е верен в структура* $\mathbf{M}$*, също са верни в* $\mathbf{M}$*.*

✓ Доказателство. Нека дизюнктът ε е тъждествено верен в структура $\mathbf{M}$, а εs е някой негов частен случай. Да допуснем, че съществува оценка v, при която този частен случай е неверен. Съгласно дефиниция 4.6.4 това означава, че всички литерали на εs са неверни в $\mathbf{M}$ при оценка v.

Съгласно лемата за субституциите 3.3.17, съществува такава оценка w, че за произволна формула, а значи и за кой да е литерал λ от ε

$$\mathbf{M} \vDash \lambda s[v] \longleftrightarrow \mathbf{M} \vDash \lambda[w]$$

Но $\mathbf{M} \vDash \lambda s[v]$ не е вярно, защото λs е литерал от εs. Значи и $\mathbf{M} \vDash \lambda[w]$ не е вярно, което означава, че литералите от ε са неверни в $\mathbf{M}$ при оценка w, следователно дизюнктът ε е неверен в $\mathbf{M}$ при оценка w. Това обаче е противоречие, защото ε е тъждествено верен в $\mathbf{M}$. ∎

Следващото следствие показва, че ако два дизюнкта са „верни“, то и тяхната резолвента е „вярна“.

✓ **4.6.20. Твърдение.** *Всяка резолвента на дизюнкти, които са верни в структура* $\mathbf{M}$*, е вярна в* $\mathbf{M}$*.*

✓ Доказателство. Следва веднага от дефиницията на либерална резолвента 4.6.14 б), току-що доказаната лема 4.6.19 и лема 4.6.18. ∎

4.6.21. Лема. *Ако* Γ *е множество от верни в структура* $\mathbf{M}$ *дизюнкти, то всички дизюнкти, които се извеждат от* Γ*, са верни в* $\mathbf{M}$*.*

✓ Доказателство. Нека $\delta_1, \delta_2, \ldots, \delta_n$ е произволен резолютивен извод от Γ. С пълна математическа индукция по i ще докажем, че дизюнктът δ_i е верен в $\mathbf{M}$.

Съгласно дефиниция 4.6.17 а), за дизюнкта δ_i има две възможности. Първата е той да бъде елемент на Γ. В този случай вече знаем, че той е верен в $\mathbf{M}$. Втората възможност е той да бъде резолвента на предходни дизюнкти в редицата. Съгласно индукционното предположение, тези предходни дизюнкти са верни в $\mathbf{M}$, и значи, съгласно лема 4.6.20, и резолвентата δ_i е вярна в $\mathbf{M}$. ∎

✓ **4.6.22. Теорема за коректност на либералната резолюция.** *Ако множеството от дизюнкти* Γ *е изпълнимо, то празният дизюнкт не е изводим от* Γ *с либерална резолюция.*

✓ Доказателство. Да допуснем, че дизюнктите от Γ имат модел $\mathbf{M}$, но въпреки това празният дизюнкт е изводим от Γ. Нека $\delta_1, \delta_2, \ldots, \delta_n$ е произволен резолютивен извод на $\square$ от Γ. Тъй като съгласно твърдение 4.6.10 $\square$ е неверен в $\mathbf{M}$, а $\delta_n = \square$, то резолютивният извод $\delta_1, \delta_2, \ldots, \delta_n$ съдържа поне един дизюнкт, който е неверен в $\mathbf{M}$. Нека j е най-малкото число, за което δ_j е неверен в $\mathbf{M}$.

Съгласно дефиниция 4.6.17 а), за дизюнкта δ_j има две възможности. Първата е той да бъде елемент на Γ. В този случай знаем, че той е верен в $\mathbf{M}$ и това е противоречие. Втората възможност е δ_j да бъде резолвента на предходни дизюнкти в редицата. Тъй като j бе най-малкото число, за което δ_j е неверен в $\mathbf{M}$, то тези предходни дизюнкти са верни в $\mathbf{M}$, и значи, съгласно лема 4.6.20, резолвентата δ_j също е вярна в $\mathbf{M}$. Така че и в този случай получаваме противоречие. ∎

Пристъпваме към доказателство на теоремата за пълнота. Тя ще гласи следното: ако от множество от дизюнкти без променливи не може да се изведе празният дизюнкт, то тогава това множество има модел. Най-напред ще разгледаме частния случай, когато работим с дизюнкти без променливи. Да забележим, че единственият частен случай на дизюнкт без променливи е самият дизюнкт, следователно при работа с дизюнкти без променливи всяка либерална резолвента е също и непосредствена резолвента.

4.6.23. Лема. *Нека* Γ *е множество от дизюнкти без променливи и* φ *е атомарна формула. Ако* $\Gamma \cup \{\{\varphi\}\} \vdash \square$ *и* $\Gamma \cup \{\{\neg\varphi\}\} \vdash \square$, *то* $\Gamma \vdash \square$.

Доказателство. Нека $\delta_1, \delta_2, \ldots, \delta_n$ е резолютивен извод на празния дизюнкт от $\Gamma \cup \{\{\varphi\}\}$. Ще дефинираме рекурсивно дизюнктите

$\delta'_1, \delta'_2, \ldots, \delta'_n$ по такъв начин, че така че винаги когато $\delta_i \neq \{\varphi\}$, да бъде вярно $\delta'_i = \delta_i$ или $\delta'_i = \delta_i \cup \{\neg\varphi\}$.

Ако $\delta_i = \{\varphi\}$, то нека δ'_i е кой да е дизюнкт от Γ.

Ако $\delta_i \in \Gamma$, то нека $\delta'_i = \delta_i$.

Ако δ_i е резолвента на предходни дизюнкти $\delta_k = \{\psi\} \cup \delta'$ и $\delta_l = \{\neg\psi\} \cup \delta''$, като $\delta_i = \delta' \cup \delta''$ и $\delta_k \neq \{\varphi\}$, то $\delta'_k = \{\psi\} \cup \delta' \underline{\cup \{\neg\varphi\}}$ и $\delta'_l = \{\neg\psi\} \cup \delta'' \underline{\cup \{\neg\varphi\}}$, като подчертаните части може и да липсват. Тогава да дефинираме $\delta'_i = \delta' \cup \delta'' \underline{\cup \{\neg\varphi\}}$, като подчертаната част липсва, ако тя липсва в δ'_k и в δ'_l. Очевидно $\delta'_i = \delta_i$ или $\delta'_i = \delta_i \cup \{\neg\varphi\}$.

Остава да дефинираме δ'_i в случая, когато δ_i е резолвента на предходни дизюнкти $\delta_k = \{\varphi\}$ и $\delta_l = \{\neg\varphi\} \cup \delta''$, като $\delta_i = \delta''$. Нека $\delta'_i = \delta'_l$. Тогава очевидно $\delta'_i = \delta_i \cup \{\neg\varphi\}$.

Във всеки един от горните четири случаи δ'_i е елемент на Γ или е резолвента на предходни дизюнкти в редицата $\delta'_1, \delta'_2, \ldots, \delta'_n$. Следователно тази редица е резолютивен извод от Γ.

Тъй като дефинирахме дизюнктите $\delta'_1, \delta'_2, \ldots, \delta'_n$ по такъв начин, че за всяко i да бъде вярно $\delta'_i = \delta_i$ или $\delta'_i = \delta_i \cup \{\neg\varphi\}$, или $\delta_i = \{\varphi\}$, но $\delta_n = \square \neq \{\varphi\}$, то $\delta'_n = \square$ или $\delta'_n = \{\neg\varphi\}$.

Ако $\delta'_n = \square$, то $\delta'_1, \delta'_2, \ldots, \delta'_n$ е резолютивен извод на $\square$ от Γ.

Ако $\delta'_n = \{\neg\varphi\}$, то нека $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k$ е резолютивен извод на $\square$ от $\Gamma \cup \{\{\neg\varphi\}\}$. Тогава

$$\delta'_1, \delta'_2, \ldots, \delta'_n, \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k$$

ще бъде резолютивен извод на $\square$ от Γ. ∎

Току що доказаната лема съдържа основната идея на доказателството на теоремата за пълнота на резолютивната изводимост. Доказателството ѝ е конструктивно, така че тя може и на практика да се използва за получаване на резолютивен извод на $\square$ от крайно множество дизюнкти без променливи. Ще илюстрираме това с пример.

4.6.24. Пример. Нека множеството Γ съдържа следните четири дизюнкта: $\{\mathtt{p}, \mathtt{q}\}$, $\{\mathtt{p}, \neg\mathtt{q}\}$, $\{\neg\mathtt{p}, \mathtt{q}\}$ и $\{\neg\mathtt{p}, \neg\mathtt{q}\}$. Това множество е неизпълнимо. Да видим как с предходната лема можем да получим резолютивен извод на $\square$ от Γ.

От $\Gamma \cup \{\{\mathtt{p}\}, \{\mathtt{q}\}\}$ тривиално се извежда $\square$:

а) $\{\neg\mathtt{p}, \neg\mathtt{q}\}$

б) $\{\mathtt{p}\}$

в) $\{\neg\mathtt{q}\}$ от а) и б)

г) $\{\mathtt{q}\}$

д) $\square$ от в) и г)

От $\Gamma \cup \{\{\mathtt{p}\}, \{\neg\mathtt{q}\}\}$ също тривиално се извежда $\square$:

а) $\{\neg\mathtt{p}, \mathtt{q}\}$

б) $\{\mathtt{p}\}$

в) $\{\mathtt{q}\}$ от а) и б)

г) $\{\neg\mathtt{q}\}$

д) $\square$ от в) и г)

Ще приложим лемата към тези два резолютивни извода, за да получим резолютивен извод на $\square$ от $\Gamma \cup \{\{\mathtt{p}\}\}$. Дизюнктите от първия от тези два резолютивни извода ще бъдат дизюнктите $\delta_1, \delta_2, \delta_3, \delta_4, \delta_5$ от доказателството на лемата. Тогава дизюнктите $\delta'_1, \delta'_2, \delta'_3, \delta'_4, \delta'_5$ от доказателството на лемата се получават от $\delta_1, \delta_2, \delta_3, \delta_4, \delta_5$ като просто пропускаме резолвентите с дизюнкта $\{\mathtt{q}\}$. Те са следните:

а) $\{\neg\mathtt{p}, \neg\mathtt{q}\}$

б) $\{\mathtt{p}\}$

в) $\{\neg\mathtt{q}\}$ от а) и б)

г) произволен дизюнкт

д) $\{\neg\mathtt{q}\}$ = в)

Виждаме, че получаваме резолютивен извод на $\{\neg\mathtt{q}\}$ от $\Gamma \cup \{\{\mathtt{p}\}\}$. Ако към този резолютивен извод „залепим“ втория от горните два резолютивни извода, ще получим следния резолютивен извод на $\square$ от $\Gamma \cup \{\{\mathtt{p}\}\}$:

а) $\{\neg\mathtt{p}, \neg\mathtt{q}\}$

б) $\{\mathtt{p}\}$

в) $\{\neg\mathtt{q}\}$ от а) и б)

г) произволен дизюнкт

д) $\{\neg\mathtt{q}\}$ = в)

е) $\{\neg\mathtt{p}, \mathtt{q}\}$

ж) $\{\mathtt{p}\}$

з) $\{\mathtt{q}\}$ от е) и ж)

и) $\{\neg\mathtt{q}\}$ = в) = д)

й) $\square$ от з) и и)

Аналогично от $\Gamma \cup \{\{\neg\mathtt{p}\}, \{\mathtt{q}\}\}$ може тривиално да изведем $\square$ по следния начин:

а) $\{\mathtt{p}, \neg\mathtt{q}\}$

б) $\{\neg \mathtt{p}\}$

в) $\{\neg \mathtt{q}\}$ от а) и б)

г) $\{\mathtt{q}\}$

д) $\square$ от в) и г)

От $\Gamma \cup \{\{\neg \mathtt{p}\}, \{\neg \mathtt{q}\}\}$ също може тривиално да изведем $\square$:

а) $\{\mathtt{p}, \mathtt{q}\}$

б) $\{\neg \mathtt{p}\}$

в) $\{\mathtt{q}\}$ от а) и б)

г) $\{\neg \mathtt{q}\}$

д) $\square$ от в) и г)

Ще приложим лемата към последните два резолютивни извода, за да получим резолютивен извод на $\square$ от $\Gamma \cup \{\{\neg \mathtt{p}\}\}$. Дизюнктите от първия от тези два резолютивни извода са $\delta_1, \delta_2, \delta_3, \delta_4, \delta_5$ от доказателството на лемата. Тогава дизюнктите $\delta'_1, \delta'_2, \delta'_3, \delta'_4, \delta'_5$ от доказателството на лемата представляват следния резолютивен извод от $\Gamma \cup \{\{\neg \mathtt{p}\}\}$:

а) $\{\mathtt{p}, \neg \mathtt{q}\}$

б) $\{\neg \mathtt{p}\}$

в) $\{\neg \mathtt{q}\}$ от а) и б)

г) произволен дизюнкт

д) $\{\neg \mathtt{q}\} =$ в)

Ако към този резолютивен извод „залепим“ втория от горните два резолютивни извода, ще получим следния резолютивен извод на $\square$ от $\Gamma \cup \{\{\neg \mathtt{p}\}\}$:

а) $\{\mathtt{p}, \neg \mathtt{q}\}$

б) $\{\neg \mathtt{p}\}$

в) $\{\neg \mathtt{q}\}$ от а) и б)

г) произволен дизюнкт

д) $\{\neg \mathtt{q}\} =$ в)

е) $\{\mathtt{p}, \mathtt{q}\}$

ж) $\{\neg \mathtt{p}\}$

з) $\{\mathtt{q}\}$ от е) и ж)

и) $\{\neg \mathtt{q}\} =$ в) $=$ д)

й) $\square$ от з) и и)

Най-накрая, ако приложим лемата към вече получените резолютивни изводи на $\square$ от $\Gamma \cup \{\{\mathtt{p}\}\}$ и от $\Gamma \cup \{\{\neg\mathtt{p}\}\}$, ще получим следния резолютивен извод на $\square$ от Γ:

а) $\{\neg\mathtt{p}, \neg\mathtt{q}\}$

б) произволен дизюнкт

в) $\{\neg\mathtt{p}, \neg\mathtt{q}\}$ = а)

г) произволен дизюнкт

д) $\{\neg\mathtt{p}, \neg\mathtt{q}\}$ = а) = в)

е) $\{\neg\mathtt{p}, \mathtt{q}\}$

ж) произволен дизюнкт

з) $\{\neg\mathtt{p}, \mathtt{q}\}$ = е)

и) $\{\neg\mathtt{p}, \neg\mathtt{q}\}$ = а) = в) = д)

й) $\{\neg\mathtt{p}\}$ от з) и и)

к) $\{\mathtt{p}, \neg\mathtt{q}\}$

л) $\{\neg\mathtt{p}\}$ = й)

м) $\{\neg\mathtt{q}\}$ от к) и л)

н) произволен дизюнкт

о) $\{\neg\mathtt{q}\}$ = м)

п) $\{\mathtt{p}, \mathtt{q}\}$

р) $\{\neg\mathtt{p}\}$ = й) = л)

с) $\{\mathtt{q}\}$ от п) и р)

т) $\{\neg\mathtt{q}\}$ = м) = о)

у) $\square$ от с) и т)

4.6.25. Ще докажем теоремата за пълнота при условие, че съществува редица (може би трансфинитна) $\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots$, съдържаща всички атомарни формули без променливи, където индексите $0, 1, 2, 3, \ldots$ са ординали. Тук няма да уточняваме какво означава „ординал“ и „трансфинитна редица“, а само ще отбележим, че всяко естествено число е пример за ординал, така че читателят може да си мисли, че горната редица е обикновена редица с индекси естествени числа. Ще отбележим без доказателство, че ако съществува редица $\mathtt{d}_0, \mathtt{d}_1, \mathtt{d}_2, \mathtt{d}_3, \ldots$, съдържаща всички символи от сигнатурата, където индексите $0, 1, 2, 3, \ldots$ са ординали, то тогава ще съществува и редица от горния вид, съдържаща всички атомарни формули без променливи. Също така ако съществува редица $\mathtt{d}_0, \mathtt{d}_1, \mathtt{d}_2, \mathtt{d}_3, \ldots$, съдържаща всички символи от сигнатурата,

където индексите $0, 1, 2, 3, \ldots$ са естествени числа, то тогава ще съществува и редица от горния вид с индекси естествени числа, съдържаща всички атомарни формули без променливи. Разбира се, когато в сигнатурата има само краен брой символи, горното условие винаги е изпълнено. Също без доказателство ще споменем, че при най-често използваната в математиката версия на аксиоматичната теория на множествата (ZFC) винаги съществува редица $\varphi_0, \varphi_1, \varphi_2, \ldots$ от искания вид.*

4.6.26. И така, да фиксираме такова множество Γ от дизюнкти без променливи, че да не е вярно $\Gamma \vdash \square$.

4.6.27. Дефиниция. Да дефинираме множествата $\Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3 \ldots$ индуктивно по следния начин:

$$\Gamma_i = \begin{cases} \Gamma_{<i} \cup \{\{\varphi_i\}\}, & \text{ако не е вярно } \Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square \\ \Gamma_{<i} \cup \{\{\neg\varphi_i\}\}, & \text{ако горното не е вярно} \end{cases}$$

където $\Gamma_{<i} = \Gamma_{i-1}$ при $i > 0$ и $\Gamma_{<0} = \Gamma$.**

Нека освен това

$$\Gamma_\infty = \bigcup_{i=0}^{\infty} \Gamma_i$$

***Забележка*:** Обикновено дефинициите, използващи разглеждане на случаи, не са конструктивни. Нека например дефинираме числото k_i по следния начин:

$$k_i = \begin{cases} 0, & \text{ако не е вярно } \Gamma_{<i} \cup \{\varphi_i\} \vdash \square \\ 1, & \text{ако горното не е вярно} \end{cases}$$

Тъй като не разполагаме с метод, посредством който можем да проверим дали е вярно $\Gamma_{<i} \cup \{\varphi_i\} \vdash \square$, то значи няма как да разберем каква е стойността на числото k_i и значи това число не е конструктивно дефинирано. Когато обаче дефинираме множества, разглеждането на случаи

*Впрочем това съществуване е неконструктивно, така че далеч не винаги може да се посочи явен пример за такава редица.

**Когато индексите може да не са естествени числа, а елементи на произволен ординал, тогава трябва да използваме следната по-обща дефиниция:

$$\Gamma_{<i} = \Gamma \cup \bigcup_{j<i} \Gamma_j$$

където считаме, че при $i = 0$ имаме $\bigcup\limits_{j<0} \Gamma_j = \varnothing$.

от горния вид е напълно конструктивно, защото винаги е възможно да пренапишем дефиницията по начин, неизползващ разглеждане на случаи. Например дефиницията на множествата Γ_i може да бъде записана по следния начин:

$$\Gamma_i = \{\, \psi \mid \psi \in \Gamma_{<i} \text{ или } (\psi = \varphi_i \text{ и не е вярно } \Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square) \text{ или } (\psi = \neg\varphi_i \text{ и не е вярно, че не е вярно } \Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square) \,\}$$

4.6.28. Лема. $\Gamma \subseteq \Gamma_0 \subseteq \Gamma_1 \subseteq \Gamma_2 \subseteq \Gamma_3 \subseteq \ldots$

Доказателство. Следва непосредствено от дефиницията на Γ_i. ∎

4.6.29. Лема. *а) Не е вярно* $\Gamma_i \vdash \square$ *за никое* i.

б) Не е вярно $\Gamma_\infty \vdash \square$.

Доказателство. **(а)** С индукция по i. Да допуснем, че $\Gamma_i \vdash \square$. Да припомним дефиницията на Γ_i:

$$\Gamma_i = \begin{cases} \Gamma_{<i} \cup \{\{\varphi_i\}\}, & \text{ако не е вярно } \Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square \\ \Gamma_{<i} \cup \{\{\neg\varphi_i\}\}, & \text{ако горното не е вярно} \end{cases} \tag{64}$$

Ако допуснем, че $\Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square$, то ще бъде верен вторият от случаите в (64), т.е. $\Gamma_i = \Gamma_{<i} \cup \{\{\neg\varphi_i\}\}$. Но ние сме допуснали, че $\Gamma_i \vdash \square$, следователно $\Gamma_{<i} \cup \{\neg\varphi_i\} \vdash \square$. Но освен това знаем и $\Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square$, затова от лема 4.6.23 получаваме $\Gamma_{<i} \vdash \square$. Последното обаче е противоречие, защото ако $i = 0$, то $\Gamma_{<i} = \Gamma$, а пък не е вярно $\Gamma \vdash \square$, и също ако $i > 0$, то $\Gamma_{<i} = \Gamma_{i-1}$, а пък $\Gamma_{i-1} \vdash \square$ не е вярно съгласно индукционното предположение.*

Следователно допускането $\Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square$ не е вярно и значи е верен първият от случаите в (64), т. е. $\Gamma_i = \Gamma_{<i} \cup \{\{\varphi_i\}\}$. Това обаче още по-бързо води до противоречие, защото от $\Gamma_i \vdash \square$ получаваме $\Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square$, което вече установихме, че не е вярно.

*Когато индексите може да не са естествени числа, а елементи на произволен ординал, то противоречието може да се получи по следния начин. Да допуснем, че съществува резолютивен извод $\delta_1, \delta_2, \ldots, \delta_n$ на $\square$ от $\Gamma_{<j}$. Всеки от дизюнктите в тази редица е елемент на $\Gamma_{<j}$ или е резолвента на предходни дизюнкти. Значи може да изберем такива дизюнкти $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_l$ измежду $\delta_1, \delta_2, \ldots, \delta_n$, че всички те са елементи на $\Gamma_{<j}$, а останалите (т. е. неизбраните) дизюнкти δ_i са резолвенти на предходни дизюнкти. Тъй като за всяко i дизюнктът ε_i е от $\Gamma_{<j}$, то за всяко i има такъв ординал k_i, че $k_i < j$ и $\varepsilon_i \in \Gamma_{k_i}$. Нека $m = \max\{k_1, k_2, \ldots, k_l\}$; тогава $m < j$. Освен това от 4.6.28 следва, че $\varepsilon_i \in \Gamma_m$ за всяко i. Следователно всеки дизюнкт от редицата $\delta_1, \delta_2, \ldots, \delta_n$ е елемент на Γ_m или е резолвента на предходни дизюнкти. Следователно тази редица е резолютивен извод на $\square$ от Γ_m, а това противоречи на индукционното предположение.

Противоречията показват, че допускането $\Gamma_i \vdash \square$ не е вярно.

(б) За да докажем, че не е вярно $\Gamma_\infty \vdash \square$, да допуснем, че $\Gamma_\infty \vdash \square$. Нека $\delta_1, \delta_2, \ldots, \delta_n$ е резолютивен извод на $\square$ от Γ_∞. Всеки от дизюнктите в тази редица е елемент на Γ_∞ или е резолвента на предходни дизюнкти. Значи може да изберем такива дизюнкти $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_l$ измежду $\delta_1, \delta_2, \ldots, \delta_n$, че всички те са елементи на Γ_∞, а останалите (т. е. неизбраните) дизюнкти δ_j са резолвенти на предходни дизюнкти. Тъй като за всяко i дизюнктът ε_i е от Γ_∞, то за всяко i има такова число k_i, че ε_i е от Γ_{k_i}. Нека $m = \max\{k_1, k_2, \ldots, k_l\}$. Тогава от 4.6.28 следва, че $\varepsilon_i \in \Gamma_m$ за всяко i. Следователно всеки дизюнкт от редицата $\delta_1, \delta_2, \ldots, \delta_n$ е елемент на Γ_m или е резолвента на предходни дизюнкти. Следователно тази редица е резолютивен извод на $\square$ от Γ_m, а това противоречи на доказаното в 4.6.29 а). ∎

4.6.30. Лема. *Нека φ е атомарна формула без променливи. Тогава:*

а) Не е възможно едновременно $\{\varphi\} \in \Gamma_\infty$ и $\{\neg\varphi\} \in \Gamma_\infty$.

б) Ако $\{\varphi\} \notin \Gamma_\infty$, то $\{\neg\varphi\} \in \Gamma_\infty$.

в) Ако $\{\neg\varphi\} \notin \Gamma_\infty$, то $\{\varphi\} \in \Gamma_\infty$.

г) $\{\varphi\} \notin \Gamma_\infty \longleftrightarrow \{\neg\varphi\} \in \Gamma_\infty$, $\quad \{\neg\varphi\} \notin \Gamma_\infty \longleftrightarrow \{\varphi\} \in \Gamma_\infty$

Доказателство. **(а)** В противен случай бихме получили празния дизюнкт като резолвента, а това противоречи на 4.6.29 б).

(б) Нека $\varphi = \varphi_i$. Щом $\{\varphi_i\} \notin \Gamma_\infty$, то $\{\varphi_i\} \notin \Gamma_i$. Да припомним дефиницията на Γ_i:

$$\Gamma_i = \begin{cases} \Gamma_{<i} \cup \{\{\varphi_i\}\}, & \text{ако не е вярно } \Gamma_{<i} \cup \{\{\varphi_i\}\} \vdash \square \\ \Gamma_{<i} \cup \{\{\neg\varphi_i\}\}, & \text{ако горното не е вярно} \end{cases} \tag{65}$$

От тази дефиниция виждаме, че не е верен първият от случаите в (65), защото не е възможно хем $\Gamma_i = \Gamma_{<i} \cup \{\{\varphi_i\}\}$, хем $\{\varphi_i\} \notin \Gamma_i$. Следователно е верен вторият случай и значи $\Gamma_i = \Gamma_{<i} \cup \{\{\neg\varphi_i\}\}$, което означава, че $\{\neg\varphi_i\} \in \Gamma_i \subseteq \Gamma_\infty$.

(в) Нека $\neg\varphi = \neg\varphi_i$. Щом $\neg\varphi_i \notin \Gamma_\infty$, то $\neg\varphi_i \notin \Gamma_i$. Поглеждайки дефиницията на Γ_i, виждаме, че не е верен вторият от двата случая в (65), защото не е възможно хем $\Gamma_i = \Gamma_{<i} \cup \{\neg\varphi_i\}$, хем $\neg\varphi_i \notin \Gamma_i$. Следователно не е вярно, че не е вярно, че не е вярно $\Gamma_{<i} \cup \{\varphi_i\} \vdash \square$, значи е верен първият от случаите в (65). Следователно $\Gamma_i = \Gamma_{<i} \cup \{\varphi_i\}$, което означава, че $\varphi_i \in \Gamma_i \subseteq \Gamma_\infty$.

(г) а) е едната посока в първата еквиваленция, а б) — другата посока. а) е едната посока във втората еквиваленция, а в) — другата посока. ■

4.6.31. Теорема за пълнота на резолютивната изводимост с дизюнкти без променливи. *Нека* Γ *е множество от от дизюнкти без променливи. Ако не е вярно* $\Gamma \vdash \square$*, то множеството* Γ *има модел. (Този модел е ербранов.)*

Доказателство. Нека Γ_∞ е дефинирано както в 4.6.27.

Нека $\mathbf{H}$ е ербранова структура, в която предикатните символи се интерпретират по следния начин:

$$\mathtt{p}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) \longleftrightarrow \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n) \in \Gamma_\infty$$

Една атомарна формула без променливи е вярна в така дефинираната структура тогава и само тогава, когато тази формула е елемент на Γ_∞. Тъй като от лема 4.6.30 г) следва, че една атомарна формула без променливи не е елемент на Γ_∞ тогава и само тогава, когато отрицанието ѝ е елемент на Γ_∞, то значи произволен литерал без променливи е верен в $\mathbf{H}$ тогава и само тогава, когато този литерал е елемент на Γ_∞.

Ще докажем, че структурата $\mathbf{H}$ е модел на Γ. За целта нека изберем произволен дизюнкт $\{\psi_1, \psi_2, \ldots, \psi_k, \neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\}$ от Γ. Трябва да докажем, че този дизюнкт е верен в $\mathbf{H}$. За целта да допуснем, че всичките му литерали са неверни в $\mathbf{H}$. От посоченото в предния абзац свойство на структурата $\mathbf{H}$ следва, че дизюнктите $\{\psi_1\}$, $\{\psi_2\}, \ldots, \{\psi_k\}$, $\{\neg\chi_1\}$, $\{\neg\chi_2\}, \ldots, \{\neg\chi_l\}$ не са елементи на Γ_∞ и значи от 4.6.30 г) получаваме, че дизюнктите $\{\neg\psi_1\}$, $\{\neg\psi_2\}, \ldots, \{\neg\psi_k\}$, $\{\chi_1\}$, $\{\chi_2\}, \ldots, \{\chi_l\}$ са елементи на Γ_∞.

Тъй като дизюнктите

$$\{\psi_1, \psi_2, \ldots, \psi_k, \neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\} \text{ и } \{\neg\psi_1\}$$

са елементи на Γ_∞ и тези два дизюнкта имат резолвента

$$\{\psi_2, \psi_3, \ldots, \psi_k, \neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\}$$

то значи тази резолвента се извежда резолютивно от Γ_∞. Дизюнктите

$$\{\psi_2, \psi_3, \ldots, \psi_k, \neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\} \text{ и } \{\neg\psi_2\}$$

се извеждат резолютивно от Γ_∞, а

$$\{\psi_3, \psi_4, \ldots, \psi_k, \neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\}$$

е тяхна резолвента, значи тази резолвента също се извежда резолютивно от Γ_∞. Продължавайки по този начин, установяваме, че

$$\{\neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\}$$

се извежда резолютивно от Γ_∞. Тъй като дизюнктите

$$\{\neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\} \text{ и } \{\chi_1\}$$

се извеждат резолютивно от Γ_∞ и имат резолвента

$$\{\neg\chi_2, \neg\chi_3, \ldots, \neg\chi_l\}$$

то тази резолвента също се извежда резолютивно от Γ_∞. Продължавайки по този начин, установяваме, че празният дизюнкт се извежда резолютивно от Γ_∞, а това е противоречие. Противоречието се дължи на допускането, че литералите на дизюнкта

$$\{\psi_1, \psi_2, \ldots, \psi_k, \neg\chi_1, \neg\chi_2, \ldots, \neg\chi_l\}$$

са неверни в $\mathbf{H}$. Щом това не е така, значи съгласно дефиниция 4.6.4 този дизюнкт е верен в $\mathbf{H}$. ∎

От тази теорема ще получим като следствие пълнотата на резолютивната изводимост за произволни дизюнкти.

4.6.32. Лема. *Нека* Γ *е множество от дизюнкти,* Γ' *е множеството от всички частни случаи без променливи на дизюнктите от* Γ *и* δ *е дизюнкт без променливи. Ако* $\Gamma' \vdash \delta$*, то* $\delta \in \Gamma'$ *или* $\Gamma \vdash \delta$*.*

Доказателство. Нека $\delta_0, \delta_1, \ldots, \delta_n$ е произволен непосредствен резолютивен извод от Γ'. С индукция по i ще докажем, че $\delta_i \in \Gamma'$ или $\Gamma \vdash \delta_i$.

Съгласно дефиницията за резолютивен извод, за δ_i има две възможности. Първата е $\delta_i \in \Gamma'$. В този случай имаме каквото искаме.

Втората възможност е δ_i да бъде резолвента на някакви предходни дизюнкти δ_m и δ_k. Съгласно индукционното предположение, $\delta_m \in \Gamma'$ или $\Gamma \vdash \delta_m$ и също така $\delta_k \in \Gamma'$ или $\Gamma \vdash \delta_k$. Ако $\delta_m \in \Gamma'$, то нека ε_m бъде дизюнкт от Γ, чийто частен случай е δ_m. Ако пък $\Gamma \vdash \delta_m$, то да положим $\varepsilon_m = \delta_m$. По аналогичен начин да дефинираме и дизюнкта ε_k. Във всеки един от тези случаи е вярно $\Gamma \vdash \varepsilon_m$ и $\Gamma \vdash \varepsilon_k$, а δ_i е резолвента на ε_m и ε_k. Следователно $\Gamma \vdash \delta_i$. ∎

4.6.33. Теорема за пълнота на либералната резолютивна изводимост. *Нека* Γ *е множество от дизюнкти. Ако не е вярно* $\Gamma \vdash \square$*, то* Γ *има модел. (Този модел е ербранов.)*

Доказателство. Нека Γ' е множеството от всички частни случаи без променливи на дизюнкти от Γ. Щом не е вярно $\Gamma \vdash \square$, то $\square \notin \Gamma$ и значи $\square \notin \Gamma'$. Ако допуснем, че $\Gamma' \vdash \square$, то от лема 4.6.32 получаваме $\Gamma \vdash \square$, а знаем, че това не е така. Значи не е вярно $\Gamma' \vdash \square$.

Щом не е вярно $\Gamma' \vdash \square$, то теоремата за пълнота 4.6.31 ни дава модел на Γ', който при това е ербранов. Съгласно твърдение 4.6.13, този ербранов модел е модел и за Γ. ∎

Икономична резолюция

Либералната резолюция е удобна за обучение, на практика обаче тя е неизползваема. Причината за това е следната: почти винаги дизюнктите имат безброй много либерални частни случаи, а поради това и безброй много либерални резолвенти. Въпреки че тези либерални резолвенти образуват алгоритмично изброимо множество и значи има начин да реализираме либералната резолюция с компютър, тази релизация ще работи само на теория, но не и на практика.

Причината, поради която методът на резолюциите е един от най-ефективните универсални методи за автоматично доказателство на теореми, е това, че измежду всичките безброй много либерални резолвенти на два дизюнкта, достатъчно е да използваме само краен брой от тях (най-често дори само една от тях). Нещо повече, благодарение на алгоритъма за унификация (вж. раздел 4.5) тези краен брой резолвенти лесно могат да бъдат намерени. В този подраздел ще видим как това може да се направи.

Обикновено един дизюнкт има безброй много частни случаи без променливи. Възниква въпросът: не е ли възможно да работим с дизюнктите по такъв начин, че всеки път, когато използваме някой дизюнкт, да може да си мислим, че все едно използваме всичките му частни случаи без променливи (дори ако те са безброй много). За бъде възможно това, е нужно така да дефинираме понятието резолвента на произволни дизюнкти, че да бъде вярно следното твърдение:

Твърдение. *Дизюнктът δ е непосредствена резолвента на някакви частни случаи без променливи на дизюнктите ε_1 и ε_2 тогава и само тогава, когато δ е частен случай без променливи на някоя резолвента на ε_1 и ε_2.*

И така, нека разгледаме как изглеждат непосредствените резолвенти на частни случаи на два дизюнкта.

4.6.34. Пример. а) Дизюнктите

$$\{\mathtt{p(x)}, \mathtt{p(a)}, \mathtt{q(x)}\} \text{ и } \{\neg\mathtt{p(y)}\}$$

имат частни случаи

$$\{\mathtt{p(b)}, \mathtt{p(a)}, \mathtt{q(b)}\} \text{ и } \{\neg\mathtt{p(b)}\}$$

Тези частни случаи имат непосредствена резолвента

$$\{\mathtt{p(a)}, \mathtt{q(b)}\}$$

б) Същите два дизюнкта имат освен това и частни случаи

$$\{\mathtt{p(a)}, \mathtt{p(a)}, \mathtt{q(a)}\} \text{ и } \{\neg\mathtt{p(a)}\}$$

Тези частни случаи имат непосредствена резолвента

$$\{\mathtt{p(a)}, \mathtt{q(a)}\}$$

в) В първия от горните два частни случая няма смисъл да записваме литерала $\mathtt{p(a)}$ два пъти. Затова тези частни случаи имат още следната непосредствена резолвента:

$$\{\mathtt{q(a)}\}$$

Разглеждайки внимателно този пример, виждаме, че във всички случаи е вярно следното твърдение:

4.6.35. Твърдение. *Дизюнктът δ е резолвента на частни случаи без променливи на дизюнктите ε' и ε'' тогава и само тогава, когато ε' и ε'' могат да се представят във вида*

$$\begin{aligned}\varepsilon' &= \{\varphi'_1, \varphi'_2, \ldots, \varphi'_n, \lambda'_1, \lambda'_2, \ldots, \lambda'_m\}\\ \varepsilon'' &= \{\neg\varphi''_1, \neg\varphi''_2, \ldots, \neg\varphi''_l, \lambda''_1, \lambda''_2, \ldots, \lambda''_k\}\end{aligned}$$

където $\varphi'_1, \ldots, \varphi'_n, \varphi''_1, \ldots, \varphi''_l$ са атомарни формули, а $\lambda'_1, \ldots, \lambda'_m, \lambda''_1, \ldots, \lambda''_k$ са литерали, и за някои ербранови оценки[*] v' и v'' е изпълнено

$$\varphi'_1 v' = \varphi'_2 v' = \cdots = \varphi'_n v' = \varphi''_1 v'' = \varphi''_2 v'' = \cdots = \varphi''_l v''$$

и

$$\delta = \{\lambda'_1 v', \lambda'_2 v', \ldots, \lambda'_m v', \lambda''_1 v'', \lambda''_2 v'', \ldots, \lambda''_k v''\}$$

[*] Вж. следващата дефиниция.

4.6.36. Дефиниция. *Ербранова оценка* е субституция, която на всяка променлива съпоставя терм без променлива.

Също както при клаузите, и при дизюнктите може да дефинираме какво значи вариант. Благодарение на вариантите става ненужно да използваме две различни ербранови оценки v' и v''. Сравнете следващата дефиниция с дефиниция 4.3.13:

4.6.37. Дефиниция. а) Да припомним, че една субституция се нарича *преименуваща*, ако заменя променливи с променливи по биективен начин.

б) Нека ε е дизюнкт. Всички дизюнкти от вида εs, където s е преименуваща субституция, се наричат *варианти* на дизюнкта ε.

4.6.38. Твърдение. *Нека $\varepsilon_1 s_1$ и $\varepsilon_2 s_2$ са варианти на дизюнктите ε_1 и ε_2, като тези варианти не съдържат общи променливи. Тогава за произволни ербранови оценки v_1 и v_2 може да се намери такава ербранова оценка v, че $(\varepsilon_1 s_1)v = \varepsilon_1 v_1$ и $(\varepsilon_2 s_2)v = \varepsilon_2 v_2$.*

Доказателство. Да означим с s_1^{-1} и s_2^{-1} обратните функции на s_1 и s_2 (те също са субституции). Нека за произволна променлива $\mathtt{z}$

$$v(\mathtt{z}) = \begin{cases} v_1(s_1^{-1}(\mathtt{z})), & \text{ако } \mathtt{z} \text{ се среща в } \varepsilon_1 s_1 \\ v_2(s_2^{-1}(\mathtt{z})), & \text{ако } \mathtt{z} \text{ се среща в } \varepsilon_2 s_2 \\ \text{произволно}, & \text{иначе} \end{cases}$$

Дефиницията на v е коректна, защото $\varepsilon_1 s_1$ и $\varepsilon_2 s_2$ не съдържат общи променливи.

Щом за всяка променлива $\mathtt{z}$ от $\varepsilon_1 s_1$ е вярно $v(\mathtt{z}) = v_1(s_1^{-1}(\mathtt{z}))$, то значи за всяка променлива $\mathtt{x}$ от ε_1 е вярно $v(s_1(\mathtt{x})) = v_1(\mathtt{x})$. Следователно $(\varepsilon_1 s_1)v = \varepsilon_1 v_1$. Аналогично се вижда, че $(\varepsilon_2 s_2)v = \varepsilon_2 v_2$. ∎

Разглеждайки твърдения 4.6.35 и 4.6.38, виждаме, че е вярно следното следствие:

4.6.39. Следствие. *Дизюнктът δ е резолвента на частни случаи без променливи на дизюнктите ε' и ε'' тогава и само тогава, когато ε' и ε'' притежават варианти без общи променливи от вида*

$$\{\varphi'_1, \varphi'_2, \ldots, \varphi'_n, \lambda'_1, \lambda'_2, \ldots, \lambda'_m\}$$
$$\{\neg\varphi''_1, \neg\varphi''_2, \ldots, \neg\varphi''_l, \lambda''_1, \lambda''_2, \ldots, \lambda''_k\}$$

където $\varphi'_1, \ldots, \varphi'_n, \varphi''_1, \ldots, \varphi''_l$ *са атомарни формули, а* $\lambda'_1, \ldots, \lambda'_m, \lambda''_1, \ldots, \lambda''_k$ *са литерали, и за някоя ербранова оценка* v *е изпълнено*

$$\varphi'_1 v = \varphi'_2 v = \cdots = \varphi'_n v = \varphi''_1 v = \varphi''_2 v = \cdots = \varphi''_l v$$

и

$$\delta = \{\lambda'_1 v, \lambda'_2 v, \ldots, \lambda'_m v, \lambda''_1 v, \lambda''_2 v, \ldots, \lambda''_k v\}$$

Това следствие ни подсеща да дадем следните дефиниции:

4.6.40. Дефиниция. а) *Дизюнкт с ограничение* е израз от вида $\langle \varepsilon \parallel \zeta \rangle$, където ε е дизюнкт, а ζ е конюнкция от равенства от вида $\tau = \sigma$, в които τ и σ са термове при сигнатурата, с която работим.

б) Ако ербрановата оценка v е такава, че

$$\tau_1 v = \sigma_1 v, \quad \tau_2 v = \sigma_2 v, \quad \ldots, \quad \tau_n v = \sigma_n v$$

то дизюнктът εv е *частен случай* на дизюнкта с ограничение

$$\langle \varepsilon \parallel \tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \cdots \,\&\, \tau_n = \sigma_n \rangle$$

4.6.41. Дефиниция. Нека дизюнктите ε' и ε'' притежават варианти без общи променливи от вида

$$\{\mathtt{p}(\tau'_{11}, \ldots, \tau'_{1k}), \mathtt{p}(\tau'_{21}, \ldots, \tau'_{2k}), \ldots, \mathtt{p}(\tau'_{m1}, \ldots, \tau'_{mk}), \lambda'_1, \lambda'_2, \ldots, \lambda'_p\}$$
$$\{\neg\mathtt{p}(\tau''_{11}, \ldots, \tau''_{1k}), \neg\mathtt{p}(\tau''_{21}, \ldots, \tau''_{2k}), \ldots, \neg\mathtt{p}(\tau''_{l1}, \ldots, \tau''_{lk}), \lambda''_1, \lambda''_2, \ldots, \lambda''_q\}$$

където $\mathtt{p}(\ldots)$ са атомарни формули, а $\lambda'_1, \ldots, \lambda'_p, \lambda''_1, \ldots, \lambda''_q$ са литерали. *Резолвента с ограничение* на ε' и ε'' ще наричаме дизюнкта с ограничение

$$\langle \lambda'_1, \lambda'_2, \ldots, \lambda'_p, \lambda''_1, \lambda''_2, \ldots, \lambda''_q \parallel \zeta \rangle$$

където ограничението ζ представлява конюнкция от вида

$$\tau'_{11} = \tau'_{21} \,\&\, \tau'_{11} = \tau'_{31} \,\&\, \ldots$$

казваща, че първите аргументи на формулите от вида $\mathtt{p}(\ldots)$ и $\neg\mathtt{p}(\ldots)$ са равни помежду си, вторите аргументи са равни помежду си, третите аргументи са равни помежду си и т. н.*

*Например ζ може да бъде конюнкцията

$$\begin{aligned}\tau'_{11} = \tau'_{21} \,\&\, \tau'_{11} = \tau'_{31} \,\&\, \cdots \,\&\, \tau'_{11} = \tau'_{m1} \\ \&\, \tau'_{12} = \tau'_{22} \,\&\, \tau'_{12} = \tau'_{32} \,\&\, \cdots \,\&\, \tau'_{12} = \tau'_{m2} \\ \&\, \cdots \,\&\, \tau'_{1k} = \tau'_{2k} \,\&\, \tau'_{1k} = \tau'_{3k} \,\&\, \cdots \,\&\, \tau'_{1k} = \tau'_{mk} \\ \&\, \tau'_{11} = \tau''_{21} \,\&\, \tau'_{11} = \tau''_{31} \,\&\, \cdots \,\&\, \tau'_{11} = \tau''_{l1} \\ \&\, \tau'_{12} = \tau''_{22} \,\&\, \tau'_{12} = \tau''_{32} \,\&\, \cdots \,\&\, \tau'_{12} = \tau''_{l2} \\ \&\, \cdots \,\&\, \tau'_{1k} = \tau''_{2k} \,\&\, \tau'_{1k} = \tau''_{3k} \,\&\, \cdots \,\&\, \tau'_{1k} = \tau''_{lk}\end{aligned}$$

Сравнявайки последните две дефиниции със следствие 4.6.39, виждаме, че те са така направени, че да бъде вярна следната лема:

4.6.42. Лема. *Дизюнктът δ е резолвента на частни случаи без променливи на дизюнктите ε' и ε'' тогава и само тогава, когато δ е частен случай на резолвента с ограничение на ε' и ε''.*

Въпреки че лема 4.6.42 има хубава формулировка, използването на резолвенти с ограничение се натъква на следното очевидно препятствие: тъй като с нейна помощ от обикновени дизюнкти получаваме дизюнкти с ограничение, то значи няма как от тези дизюнкти с ограничение да получим нови резолвенти. Следователно или трябва да се научим да правим резолвенти с ограничение, ако това, с което разполагаме, са пак дизюнкти с ограничение, или трябва да се научим да елиминираме ограничението, така че да получим обикновен дизюнкт. Вторият вариант е за предпочитане и той, за щастие, се оказва възможен.

Да си припомним, че благодарение на алгоритъма за унификация за произволно ограничение от равенства между термове можем да установим дали то има решение и ако има, то можем с помощта на еквивалентни преобразувания да го сведем към вида

$$\mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \cdots \,\&\, \mathtt{x}_n = \tau_n$$

4.6.43. Дефиниция. а) Нека ни е даден дизюнктът с ограничение $\langle \varepsilon \parallel \zeta \rangle$. Ако използвайки алгоритъма за унификация сведем ограничението до вида

$$\mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \cdots \,\&\, \mathtt{x} = \tau_n$$

то дизюнктът

$$\varepsilon[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n := \tau_1, \tau_2, \ldots, \tau_n]$$

ще наричаме резултат от елиминацията на ограничението в дадения дизюнкт с ограничение.

б) Ако дизюнктът с ограничение $\langle \varepsilon \parallel \zeta \rangle$ е резолвента с ограничение на ε_1 и ε_2 и δ е резултатът от елиминацията на ограничението в $\langle \varepsilon \parallel \zeta \rangle$, то дизюнктът δ ще наричаме *икономична резолвента* на ε_1 и ε_2.

4.6.44. Пример. Да разгледаме дизюнктите от пример 4.6.34

$$\{\mathtt{p}(\mathtt{x}), \mathtt{p}(\mathtt{a}), \mathtt{q}(\mathtt{x})\} \text{ и } \{\neg\mathtt{p}(\mathtt{y})\}$$

Тъй като в дефиницията за резолвента с ограничение се говори за варианти без общи променливи, да се уговорим, че на променливите от първия дизюнкт винаги ще слагаме примове, а на променливите на втория дизюнкт — секонди:

$$\{\mathtt{p}(\mathtt{x}'), \mathtt{p}(\mathtt{a}), \mathtt{q}(\mathtt{x}')\} \text{ и } \{\neg\mathtt{p}(\mathtt{y}'')\}$$

Една резолвента с ограничение е следната:

$$\langle\{\mathtt{p}(\mathtt{a}), \mathtt{q}(\mathtt{x}')\} \parallel \mathtt{x}' = \mathtt{y}''\rangle$$

Тъй като ограничението вече е решено, остава само да приложим субституцията $[\mathtt{x}' \coloneqq \mathtt{y}'']$, за да получим резолвентата

$$\{\mathtt{p}(\mathtt{a}), \mathtt{q}(\mathtt{y}'')\}$$

Друга резолвента с ограничение на горните дизюнкти е следната:

$$\langle\{\mathtt{q}(\mathtt{x}')\} \parallel \mathtt{a} = \mathtt{x}' \,\&\, \mathtt{x}' = \mathtt{y}''\rangle$$

Прилагайки алгоритъма за унификация към ограничението $\mathtt{a} = \mathtt{x}'\&\mathtt{x}' = \mathtt{y}''$, получаваме $\mathtt{x}' = \mathtt{a} \,\&\, \mathtt{y}'' = \mathtt{a}$. Следователно трябва да приложим субституцията $[\mathtt{x}', \mathtt{y}'' \coloneqq \mathtt{a}, \mathtt{a}]$ към $\{\mathtt{q}(\mathtt{x}')\}$, за да получим резолвентата

$$\{\mathtt{q}(\mathtt{a})\}$$

4.6.45. Лема. *Ако ε' е резултатът от елиминацията на ограничението в дизюнкта $\langle\varepsilon \parallel \zeta\rangle$, то ε' и $\langle\varepsilon \parallel \zeta\rangle$ имат едни и същи частни случаи без променливи.*

Доказателство. Нека алгоритъмът за унификация свежда ограничението ζ до

$$\mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \cdots \,\&\, \mathtt{x} = \tau_n$$

и

$$\varepsilon' = \varepsilon[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n]$$

По дефиниция всички частни случаи без променливи на $\langle\varepsilon\|\zeta\rangle$ имат вида εv, където v е ербранова оценка, която е решение на ограничението ζ. Но v е решение на ζ тогава и само тогава, когато

$$v(\mathtt{x}_1) = \tau_1 v, \;\; v(\mathtt{x}_2) = \tau_2 v, \;\; \ldots, \;\; v(\mathtt{x}_n) = \tau_n v$$

Следователно

$$\begin{aligned}\varepsilon v &= \varepsilon[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1 v, \tau_2 v, \ldots, \tau_n v | v] \\ &= \varepsilon[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n] v = \varepsilon' v\end{aligned}$$

Последното показва, че частните случаи без променливи на $\langle \varepsilon \parallel \zeta \rangle$ са частни случаи и на ε'.

Остава да докажем, че всеки частен случай без променливи на ε' е частен случай и на $\langle \varepsilon \parallel \zeta \rangle$. Нека $\varepsilon' w$ е произволен частен случай без променливи на ε'. Тогава

$$\begin{aligned}\varepsilon' w &= \varepsilon[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n] w \\ &= \varepsilon[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1 w, \tau_2 w, \ldots, \tau_n w | w] = \varepsilon u\end{aligned}$$

където

$$u = [\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1 w, \tau_2 w, \ldots, \tau_n w | w]$$

От дефиницията на ербрановата оценка u следва, че

$$u(\mathtt{x}_1) = \tau_1 w, \quad u(\mathtt{x}_2) = \tau_2 w, \quad \ldots, \quad u(\mathtt{x}_n) = \tau_n w \tag{66}$$

От друга страна алгоритъмът за унификация гарантира, че променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ не се срещат в термовете $\tau_1, \tau_2, \ldots, \tau_n$, и значи

$$\tau_1 u = \tau_1 w, \quad \tau_2 u = \tau_2 w, \quad \ldots, \quad \tau_n u = \tau_n w \tag{67}$$

От (66) и (67) следва

$$u(\mathtt{x}_1) = \tau_1 u, \quad u(\mathtt{x}_2) = \tau_2 u, \quad \ldots, \quad u(\mathtt{x}_n) = \tau_n u$$

и значи u е решение на ограничението ζ. Следователно частният случай $\varepsilon' w$, за който установихме, че е равен на εu, е частен случай на $\langle \varepsilon \| \zeta \rangle$. ∎

4.6.46. Следствие. *Дизюнктът δ е резолвента на частни случаи без променливи на дизюнктите ε' и ε'' тогава и само тогава, когато δ е частен случай без променливи на икономична резолвента на ε' и ε''.*

Доказателство. Следва от предната лема и лема 4.6.42. ∎

4.6.47. Дефиниция. а) Нека Γ е множество от дизюнкти. *Икономичен резолютивен извод* на дизюнкта δ от Γ ще наричаме редица от дизюнкти

$$\delta_0, \delta_1, \delta_2, \ldots, \delta_n$$

в която $\delta_n = \delta$ и за всеки от дизюнктите в редицата е вярно поне едно от следните две условия: 1) че е елемент на Γ или 2) че е икономична резолвента на дизюнкти, които се срещат по-рано в редицата.

б) Дизюнктът δ се *извежда* от Γ с икономична резолюция, ако съществува икономичен резолютивен извод на δ от Γ.

4.6.48. Лема. *Нека* Γ *е множество от дизюнкти и* Γ' *е множеството от всички частни случаи без променливи на дизюнктите от* Γ. *Празният дизюнкт се извежда с икономична резолюция от* Γ *тогава и само тогава, когато той се извежда от* Γ'.

Доказателство. Най-напред да докажем, че ако празният дизюнкт се извежда от Γ', то той се извежда и от Γ. Да допуснем, че

$$\delta_0, \delta_2, \ldots, \delta_n$$

е резолютивен извод на $\square$ от Γ'. Да построим по следния начин такава редица

$$\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_n$$

че за всяко i δ_i е частен случай на ε_i. Когато δ_i е елемент на Γ', то нека ε_i такъв елемент на Γ, че δ_i е частен случай на ε_i. Когато пък δ_i е непосредствена резолвента на предходни дизюнкти δ_m и δ_k, то от следствие 4.6.46 получаваме такъв дизюнкт ε_i, че ε_i е икономична резолвента на ε_m и ε_k, а δ_i е частен случай на ε_i.

Редицата $\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_n$ е резолютивен извод. Тъй като $\delta_n = \square$ и δ_n е частен случай на ε_n, то $\varepsilon_n = \square$.

Сега да видим обратното: че ако празният дизюнкт се извежда от Γ с икономична резолюция, то той се извежда и от Γ'. За целта ще докажем, че ако някакъв дизюнкт ε се извежда от Γ с икономична резолюция, то всички негови частни случаи без променливи се извеждат от Γ'. Това ще докажем с пълна математическа индукция по дължината на резолютивния извод на ε. И така, нека

$$\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$$

е икономичен резолютивен извод на ε от Γ. За ε_n има две възможности. Първата е $\varepsilon = \varepsilon_n \in \Gamma$. В този случай всички частни случаи без променливи на ε са елементи на Γ' и значи са изводими от Γ'. Втората възможност за дизюнкта ε_n е той да бъде резолвента на някои предходни дизюнкти ε_m и ε_k. Нека δ е произволен частен случай без променливи на ε. Съгласно следствие 4.6.46, съществуват такива частни случаи δ' и δ'' съответно на ε_m и ε_k, че δ е непосредствена резолвента на δ' и δ''. Тъй като ε_m и ε_k притежават резолютивни изводи с дължини съответно m и k и $m < n, k < n$, то съгласно индукционното предположение,

съществуват резолютивни изводи на δ' и δ'' от Γ':

$$\delta'_1, \delta'_2, \ldots, \delta'_l = \delta'$$
$$\delta''_1, \delta''_2, \ldots, \delta''_j = \delta''$$

Тогава

$$\delta'_1, \delta'_2, \ldots, \delta'_l, \delta''_1, \delta''_2, \ldots, \delta''_j, \delta$$

ще бъде резолютивен извод на δ от Γ'. ■

4.6.49. Теорема за коректност и пълнота на резолютивната изводимост. *Едно множество от дизюнкти* Γ *е изпълнимо тогава и само тогава, когато празният дизюнкт не е изводим от* Γ *посредством икономична резолютивна изводимост.*

Доказателство. Съгласно „малката“ теорема на Ербран за дизюнкти 4.6.7, множеството Γ е изпълнимо тогава и само тогава, когато то има ербранов модел. От твърдение 4.6.13 следва, че един дизюнкт е тъждествено верен в ербранова структура тогава и само тогава, когато в ербрановата структура са верни всички негови частни случаи. Следователно множеството Γ е изпълнимо тогава и само тогава, когато множеството Γ' от всички негови частни случаи има ербранов модел. Отново съгласно „малката“ теорема на Ербран, последното е така тогава и само тогава, когато Γ' има модел. Съгласно теоремите за коректност и пълнота на либералната резолюция (4.6.22 и 4.6.33), Γ' има модел тогава и само тогава, когато празният дизюнкт е изводим от Γ'. Съгласно лема 4.6.48, това е така тогава и само тогава, когато празният дизюнкт е изводим от Γ с икономична резолюция. ■

Приложение А

Термове

А.1. Термовете в пролог

Синтаксис

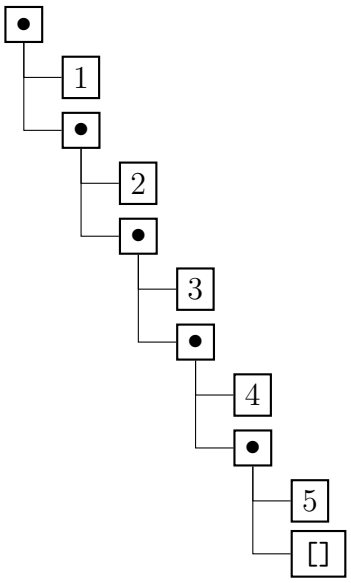
А.1.1. Езикът пролог използва синтаксис, подобен на математическия, със следните по-важни различия:

Първо, функционалните символи и символите за константи на пролог могат да се претоварват. Така например `f(f(f,f))` е правилен терм. В него първият символ `f` е едноместен, вторият — двуместен, а третият и четвъртият — символи за константи. Въпреки че са означени по един и същ начин, пролог интерпретира тези символи като напълно различни един от друг.[**]

Второ, пролог не използва математическата уговорка, съгласно която напр. `b` трябва да бъде символ за константа, а `x` — променлива. Вместо това, пролог използва следното правило: ако името започва с главна буква, то тогава то е променлива, а ако започва с малка буква, то е символ за константа или функционален символ.[***] Например `x(C)` е правилно построен терм, в който `x` е едноместен функционален символ, а `C` е променлива.

[**]За избягване на недоразумения, в документацията на пролог арността се обозначава с наклонена черта /. Например `f/2` е двуместният функционален символ `f`, а `c/0` — символът за константа `c`.

[***]Броят на аргументите показва дали е символ за константа или функционален символ.



Фиг. 17. Синтактично дърво за списъка `[1,2,3,4,5]`

Трето, както във всеки нормален език за програмиране, имената в пролог могат да бъдат многобуквени. Например `abc(abc,Abc)` е терм, в който първото `abc` е двуместният функционален символ `abc`, второто `abc` е символът за константа `abc`, а `Abc` е променлива.

В езика пролог не се прави разлика между символи за константи, функционални символи и предикатни символи, а атомарните формули са просто вид термове.

Списъци

Списъците са една от най-важните структури данни в програмирането. Ако представим списъка `[1,2,3,4,5]` във вид на свързан списък, в паметта на компютъра ще се образува структура, която донякъде наподобява синтактичното дърво, показано във фигура 17. Като се вземе предвид това, че термовете са универсален тип данни, фактът, че можем да използваме синтактични дървета (т.е. термове), за да представяме списъци, никак не е изненадващ.

Следователно ако искаме да използваме списъка `[1,2,3,4,5]` на пролог, вместо него можем да използваме терма

```
•(1,•(2,•(3,•(4,•(5,[])))))
```

В този терм `[]` е символ за константа, представящ списъка с нула елементи (т. н. *празен списък*),* а $\bullet$ е двуместен функционален символ като смисълът на $\bullet(A, X)$ е списък, чийто пръв елемент е A, а X е списък от всички елементи без първия.

Първият елемент на даден списък се нарича *глава* на списъка, а списъкът от всички елементи без първия — *опашка*. Например 1 е глава на списъка $\bullet$`(1,`$\bullet$`(2,`$\bullet$`(3,`$\bullet$`(4,`$\bullet$`(5,[])))))`, а $\bullet$`(2,`$\bullet$`(3,`$\bullet$`(4,`$\bullet$`(5,[]))))` — опашка.

Използването на този синтаксис за работа със списъци е възможно, но неудобно. Затова в пролог е предвидено следното улеснение: вместо да пишем списъци като $\bullet$`(1,`$\bullet$`(2,`$\bullet$`(3,`$\bullet$`(4,`$\bullet$`(5,[])))))`, можем да използваме далеч по-четливия запис `[1,2,3,4,5]`. Този начин за записване на списъци обаче не е нищо повече от синтактична захар, тъй като не добавя никакви нови възможности към езика, а само прави програмите по-кратки и по-четливи, улеснявайки по този начин живота на програмистите.

Вместо да пишем $\bullet(\alpha, \chi)$, пролог позволява да използваме записа `[`α`|`χ`]`. Тук α е главата, т.е. първият елемент на `[`α`|`χ`]`, а χ — опашката. Освен това можем да пишем и изрази като `[`$\alpha_1, \alpha_2, \alpha_3$`|`$\chi$`]` вместо $\bullet(\alpha_1, \bullet(\alpha_2, \bullet(\alpha_3, \chi)))$. Тук α_1 е първият елемент на `[`$\alpha_1, \alpha_2, \alpha_3$`|`$\chi$`]`, α_2 — вторият елемент, α_3 — третият, а χ е списъкът от всички елементи от четвъртия нататък.**

Ще казваме, че даден запис на спъсък на пролог е *ненормален*, ако в него скобата `[` се среща непосредствено след вертикална черта `|`. В противен случай записът е *нормален*. Винаги има начин да трансформираме един ненормален запис на списък в нормален. Например списъкът `[1|[2|[]]]` е равен на `[1,2]`, а списъкът `[[1,2]|[3,4]]` е равен на `[[1,2],3,4]`. Нарисувайте синтактични дървета, за да се убедите, че това е така.***

***Задача* 51:** Уверете се, че термът

$\bullet$`(`$\bullet$`(`$\bullet$`(1,[]),`$\bullet$`(2,[])),`$\bullet$`(`$\bullet$`(3,`$\bullet$`(4,[])),`$\bullet$`(5,`$\bullet$`(6,[]))))`

*Тук се вижда разликата между „константа“ и „символ за константа“. Празният списък е константа (в случая нещо, което може би се намира някъде в паметта на компютъра), а `[]` е символ (т.е. означение) за тази константа.

**Ако списъкът съдържа само трите елемента α_1, α_2 и α_3, то χ е празният списък.

***Нормалният запис на списъците е много по-четлив, така че никога няма смисъл да записваме списъците по ненормален начин. Затова, когато на писмения изпит някоя програма на пролог съдържа ненормални списъци, проверяващите започват да стават подозрителни.

е същият като списъка с ненормален запис

```
[[[1|[]]|[2|[]]]|[[3|[4|[]]]|[5|[6|[]]]]]
```

Нарисувайте синтактично дърво за този списък. Как можем да го запишем по нормален начин? Забележете колко по-четлив е нормалният запис.

Оператори

> *Операторите се използват с цел да направят програмите по-четливи. Хората са доста гъвкави и могат да се приспособят към странни среди — например те могат да свикнат да четат програми на* ЛИСП *и* ФОРТРАН. *Въпреки това, ние считаме, че синтаксисът е важен; силата на добрата нотация е добре известна в математиката. Съществена част от хубавия синтаксис на пролог е възможността да се описват и използват оператори.*
>
> Лион Стерлинг *и* Ехуд Шапиро *[18]*

Да си припомним, че когато математиците напишат терм във вида $3/4+(2-4*5)$, те всъщност имат предвид терма $+(/(3,4),-(2,*(4,5)))$. И пролог прави същото — изразът `3/4+(2-4*5)` ще бъде изтълкуван така, сякаш вместо това сме записали `+(/(3,4),-(2,*(4,5)))`.*

Нещо интересно обаче, което отличава пролог от почти всички останали езици за програмиране, е това, че пролог позволява на програмиста да дефинира и много други оператори. Използвайки тази възможност, на пролог могат да се дефинират езици с доста сложен синтаксис. Например в приложение Б е дадена пълната реализация на един императивен език за програмиране, наречен ИМПЕРАТОР, като това е така направено, че ИМПЕРАТОР да стане част от езика пролог. Ето как например може да дефинираме предикат, за пресмятането на факториела на дадено число:

```
% по дадено естествено число N записва в F неговия факториел
факториел(N,F) :- императоp
   f := 1;
   for i in 2..N loop
```

*Личи си, че пролог е измислен във Франция, нали?

```
    f := f * i;
F <== f.
```

Дефинирането на нови оператори на пролог става посредством вградения предикат `op`. Например след изпълнението на*

```
:- op(700,xf, [е_животно]).
:- op(700,xfx,[има]).
:- op(600,fy, [бащата_на]).
```

пролог ще интерпретира изразите

```
бащата_на лиско
лиско е_животно
бащата_на лиско е_животно
бащата_на бащата_на лиско има козина
```

все едно, че сме написали термовете

```
бащата_на(лиско)
е_животно(лиско)
е_животно(бащата_на(лиско))
има(бащата_на(бащата_на(лиско)), козина)
```

Тук числото, което даваме като пръв аргумент на `op`, е приоритетът на оператора като колкото по-малко е числото, толкова по-голям е приоритетът. Да забележим, че ако на оператора `бащата_на` дадем приоритет 800 вместо 600, то пролог ще интерпретира третия от горните изрази като

```
бащата_на(е_животно(лиско))
```

което, разбира се, е правилен терм, но лишен от смисъл.

Вторият аргумент на `op` определя типа на оператора. Съществуват следните типове:

fx Едноместни префиксни оператори, в които аргументът не може да има същият приоритет. Например ако дефинираме оператор `вали` от тип `fx`, то изразът `вали дъжд` ще се интерпретира като `вали(дъжд)`, а `вали вали дъжд` ще бъде невалиден израз.

fy Едноместни префиксни оператори, в които аргументът може да има същият приоритет. Например `бащата_на бащата_на лиско`.

*Не всяка реализация на пролог позволява директното използване на кирилица във функционалните символи, символите за константи и променливите. суи-пролог е една от най-популярните реализации на пролог и тя позволява това. Вижте `http://www.swi-prolog.org`. Кирилица може да се използва и при строубери пролог. Вижте `http://dobrev.com`.

оператор	*тип*	*приоритет*
унарни + и -	`fy`	200
^	`xfy`	200
* и /	`yfx`	400
бинарни + и -	`yfx`	500
= и <	`xfx`	700
,	`xfy`	1000
;	`xfy`	1100
:- и ?-	`xfx`	1200

Таблица 5. Приоритет и тип на някои от предефинираните оператори на пролог

xf *и* ***yf*** Аналогично на `fx` и `fy`, но за постфиксни оператори. Например `лиско е_животно`.

yfx Лявоасоциативни инфиксни оператори. Например изразът `2+3+4` се интерпретира като `(2+3)+4`.

xfy Дясноасоциативни инфиксни оператори. Например изразът `2^3^4` се интерпретира като `2^(3^4)`.

xfx Двуместни инфиксни оператори, при които нито отляво, нито отдясно се позволяват оператори със същия приоритет. Например изразите 2=3=4 и 2<3<4 са недопустими, защото операторите = и < са от тип `xfx`.

При определяне на приоритета на операторите е добре да се имат предвид приоритетите на операторите, които в пролог са предефинирани. По-важните от тях са дадени в таблица 5.

Операторът запетая в пролог е особен, защото се използва хем като инфиксен оператор (като смисълът му е конюнкция), хем като разделител между аргументите. Например ако искате да кажете, че не е вярно, че „5=5 и 3=4“ и напишете израза `not(5=5,3=4)`, пролог ще се оплаче, че не е ясен смисълът на `not`, когато се използва с два аргумента. Проблемът се решава с една двойка допълнителни скоби: `not((5=5,3=4))`.

***Задача* 52:** В пролог `not` е обикновен функционален символ, а не оператор. В следствие на това трябва да пишете `not(5=4)` вместо `not 5=4` и `not((5=5,3=4))` вместо `not (5=5,3=4)`. Обаче, ако дефинирате `not` като оператор, изразите `not 5=4` и `not (5=5,3=4)` ще станат коректни.* Как можете да направите това?

*Забележете интервала между `not` и отворената скоба.

***Задача* 53:** Дефинирайте на пролог оператори `и`, `или`, `не`, благодарение на които ще можем да програмираме на български например така:

```
ремонт(Кола) :-
   не добри_спирачки(Кола) или
   шумна(Кола) или пуши(Кола) или
   външна_повреда(Кола,Повреда) и не драскотина(Повреда) или
   течове(Кола, Количество) и Количество > 194.8.
```

А.2. Термовете във функционалните езици

Функционалните езици образуват две големи семейства — семейството на диалектите на ЛИСП, към което спада напр. скийм, и семейството на статичнотипизираните функционални езици като хаскел, о-камъл, скала и кложър.

Също както при пролог в общи линии всички неща са термове, така на ЛИСП пък всички неща са списъци. За списъците се използва следният синтаксис: $(\alpha_1\ \alpha_2\ \ldots\ \alpha_n)$. С други думи забелязват се следните разлики спрямо синтаксиса, използван от пролог: списъците се заграждат в кръгли, а не в квадратни скоби и между елементите не се поставят запетаи, а интервали.

Въпреки че ЛИСП няма директна поддръжка за термове, оказва се, че никак не е трудно да използваме списъци, за да записваме термове — това става като вместо терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ използваме списъка $(\mathtt{f}\ \tau_1\ \tau_2\ \ldots\ \tau_n)$. С други думи първият елемент на списъка показва кой е функционалният символ, а следващите елементи са аргументите. Например термът $\mathtt{f}(\mathtt{g}(\mathtt{a}, \mathtt{b}), \mathtt{c})$ може да бъде записан на ЛИСП така:

```
(f (g a b) c)
```

В раздел 2.6 видяхме, че променливите — както в математиката, така и в програмирането — биват два вида: свободни и свързани. Термовете във функционалните езици, и в частност при ЛИСП и диалектите му, не могат да съдържат свободни променливи.

За сметка на това обаче диалектите на ЛИСП позволяват да се използват т. н. ламбда-термове, които съдържат свързани променливи. Например на скийм

```
(lambda (f g) (lambda (x) (f (g x))
```

е ламбда-терм, съдържащ трите свързани променливи `f`, `g` и `x`. Този ламбда-терм представя функцията, която взема като аргументи две

функции `f` и `g` и връща като стойност тяхната композиция (което, разбира се, също е функция). Математиците[*] записват този ламбда-терм така: $\lambda fg.\lambda x.f(gx)$.

На пролог свързани променливи няма.[**]

* * *

При другата голяма група от функционални езици — тази на статичнотипизираните езици като хаскел, о-камъл, скала и кложър — в следствие на строгата типизация не е удобно „да се хитрува“ като се използват списъци вместо термове. Затова тези езици имат по-непосредствена поддръжката за термове, отколкото ЛИСП.

В много от статичнотипизираните функционални езици за термовете се използва същият синтаксис, както и при ЛИСП, с тази разлика, че не е задължително да пишем най-външните кръгли скоби (въпреки това в примерите по-долу най-външните скоби са слагани). Например `(f` α_1 α_2 `...`α_n`)` е термът с функционален символ `f` и аргументи $\alpha_1, \alpha_2, \ldots, \alpha_n$.

Да допуснем, че искаме да разполагаме с функционални символи двуместен `Point`, двуместен `Rectangle`, едноместен `Circle` и триместен `Positioned_figure`.

Смисълът на `(Point` x y`)` е точка с координати (x, y).

Смисълът на `(Rectangle` a b`)` е геометричната фигура правоъгълник със страни a и b като позицията на правоъгълника не е фиксирана.

Смисълът на `(Circle` r`)` е геометричната фигура окръжност с радиус r като позицията на окръжността не е фиксирана.

Смисълът на `(Positioned_figure` A φ σ`)` е геометричната фигура σ (която може да бъде правоъгълник или окръжност) поставена в точката A и завъртяна на ъгъл φ.[***]

Например

```
(Positioned_figure (Point 1 5) 30 (Rectangle 3 2))
```

представлява правоъгълник със страни 3 и 2, поставен в точка с координати $(1, 5)$ и завъртян на ъгъл 30.[****]

[*] И по-точно Аланзо Чърч през 1932 г.

[**] Свързани променливи има например в езика ламбда-пролог.

[***] Разбира се, да въртим окръжност няма смисъл. Обаче в един по-реалистичен пример освен правоъгълника със сигурност би имало и много други форми, които не са ротационно инвариантни.

[****] Каквото и да значи изразът „завъртян на ъгъл 30“. Много самолети и ракети са падали заради това, че една част от програмата използва радиани, а друга градуси или едната метри, а другата футове.

Преди да можем да дефинираме тези функционални символи, трябва да определим какви типове използват те. Забелязваме, че се използват общо четири типа:

`Float` Т.е. реално число. Този тип почти винаги е вграден в езиците и не е нужно да го дефинираме.

`Point_type` Стойността на функционалния символ `Point` и първият аргумент на `Positioned_figure` има този тип.

`Figure` Стойността на `Rectangle` и `Circle` има този тип.

`Graphic_object` Това е типът на стойността на `Positioned_figure`.

След като сме решили кои са функционалните символи и какви са техните типове, можем да ги дефинираме формално. Забележете, че дефиницията на функционалните символи е част от дефиницията на типовете.

Дефиницията на хаскел изглежда така:

```
data Point_type = Point Float Float
data Figure =
   | Rectangle Float Float
   | Circle Float
data Graphic_object = Positioned_figure Point_type Float Figure
```

Същата дефиницията на о-камъл изглежда така (на о-камъл типовете се пишат с малки букви):

```
type point_type = Point of float * float
type figure =
   Rectangle of float * float
   | Circle of float
type graphic_object =
   Positioned_figure of point_type * float * figure
```

Приложение Б

Реализация на езика за програмиране император

В това приложение ще видим как можем да напишем на пролог интерпретатор на прост императивен език за програмиране и то така, че този език да стана част от пролог. Понеже всеки език за програмиране трябва да си има име, нека дадем на нашия език името ИМПЕРАТОР.

Най-напред ще определим синтаксиса на ИМПЕРАТОР. Всяка програма на ИМПЕРАТОР ще се състои просто от ключовата дума `император`, следвана от списък от команди, разделени със символа точка и запетая. Всяка команда ще има един от следните пет вида.

Първо, оператор за присвояване. Отляво трябва да стои идентификатор, който си го мислим като име на променлива на ИМПЕРАТОР:

```
променлива := израз
```

Второ, присвояване на стойност на променлива на пролог. Отляво трябва да стои име на променлива на пролог, която все още не е получила стойност:

```
променлива <== израз
```

Трето, изпълнение на предикат на пролог (също както в програма на пролог можем да изпълняваме код, написан на ИМПЕРАТОР, така и в програма на ИМПЕРАТОР можем да извикваме предикати на пролог):

```
пролог p(T1,T2,...,Tn)
```

Четвърто, оператор за цикъл while. Ще считаме, че цикълът се изпълнява докато стойността на контролиращия израз е различна от нула:

```
while израз loop (
   оператор 1;
   оператор 2;
   ...
   оператор n
)
```

Пето, оператор за цикъл for:

```
for i in A..B loop (
   оператор 1;
   оператор 2;
   ...
   оператор n
)
```

Шесто, условен оператор. Приемаме, че изразът е истина, ако стойността му е различна от нула:

```
if израз then (
   оператор 1;
   оператор 2;
   ...
   оператор n
)
```

или

```
if израз then (
   оператор 1;
   оператор 2;
   ...
   оператор n
) else (
   оператор 1;
   оператор 2;
   ...
   оператор m
)
```

Използвайки този език, ето как можем да дефинираме на пролог предикат за пресмятане на факториела на едно число:

```
% по дадено естествено число N записва в F неговия факториел
факториел(N,F) :- императ
   f := 1;
   for i in 2..N loop (
      f := f * i
   );
   F <== f.
```

В последния ред от тази програма присвояваме на променливата с име F на пролог стойността, която в този момент променливата с име f на ИМПЕРАТОР има.

Пълната реализация на ИМПЕРАТОР е дадена малко по-нататък. Ще коментираме някои по-трудни за разбиране неща. Първо, да видим как е реализиран синтаксисът на ИМПЕРАТОР.

Операторите `император`, `:=`, `<==` и `пролог` са дефинирани така:

```
:- op(1150, fx, [император]).
:- op(990, xfx, [':=', '<==']).
:- op(990, fx, [пролог]).
```

Тук приоритетът 1150 е подбран така, че да бъде по-малко число (т.е. по-голям приоритет) от приоритета на `:-`, но по-голямо число от приоритета на оператора точка и запетая. Първото ни гарантира, че няма нужда да ограждаме програмата на ИМПЕРАТОР в скоби така: `p :- (император ...)`. Второто ни позволява да не слагаме скоби след `император` само заради това, че в програмата се съдържат команди, разделени с точка и запетая ето така: `p :- император (... ; ...)`.*

Приоритетът 990 е подбран така, че бъде по-голямо число (т.е. по-малък приоритет) от приоритетите на всички аритметични операции в пролог, но по-малко число, от приоритета на оператора точка и запетая. Първото ни гарантира, че няма нужда да пишем скоби в израза след `:=` и `<==`, например `i := (1+2)`. Второто пък означава, че няма нужда да пишем скоби около самия оператор за присвояване, например `(i:=1);(f:=1)`.

Синтаксисът на оператора за цикъл while на ИМПЕРАТОР използва две различни ключови думи — `while` и `loop`. Всяка една от тях трябва

*Обаче скоби около програмата на ИМПЕРАТОР все пак трябва да се поставят, ако тя не е единственото нещо, което се прави в предиката на пролог:

```
p(X) :- q(X),
        (император
            тук е програмата на ИМПЕРАТОР
        ),
        r(X).
```

да бъде дефинирана като префиксен, постфиксен или инфиксен оператор на пролог със свой собствен приоритет. Един начин това да бъде направено е да дефинираме `while` като префиксен оператор, а `loop` — като инфиксен с два аргумента. Ще дадем на `while` по-голям приоритет, отколкото на `loop`, така че в израза

`while` израз `loop (` команди `)`

пролог да постави скобите по следния начин:

`(while` израз `) loop (` команди `)`

Записан като терм същият цикъл изглежда така:

`loop(while(`израз`),` команди`)`

Ще изберем така приоритетите на `while` и `loop`, че те да бъдат по-големи от приоритета на оператора точка и запетая, но по-малки от приоритетите на операторите `:=`, `<==` и `пролог`:

```
:- op(994, fx, [while]).
:- op(995, xfx, [loop]).
```

Операторът за цикъл for използва операторите `for`, `in`, `..` и `loop`. Така ще ги дефинираме, че в израза

`for` i `in` a`..`b `loop (` команди `)`

пролог да постави скобите по следния начин:

`(for (`i `in (`a`..`b`))) loop (` команди `)`

Записан като терм същият цикъл изглежда така:

`loop(for(in(i,(a..b))),` команди`)`

Ще изберем така приоритетите на `for`, `in` и `..`, че те да бъдат по-големи от приоритета на `loop` и по-малки от приоритетите на аритметичните оператори:

```
:- op(994, fx, [for]).
:- op(993, xfx, [in]).
:- op(992, xfx, ['..']).
```

Условният оператор на ИМПЕРАТОР използва две или три ключови думи — `if`, `then` и може би също `else`:

```
:- op(995, fx, [if]).
:- op(993, xfx, [then]).
:- op(994, xfx, [else]).
```

Така избраните приоритети означават, че пролог ще интерпретира изразите

```
if израз then ( команди )
if израз then ( команди1 ) else ( команди2 )
```

като термовете

```
if(then(израз,команди))
if(else(then(израз,команди1),команди2)
```

За разлика от пролог, на ИМПЕРАТОР променливите могат да си променят стойността. Това означава, че не можем да помним стойността на променливите на ИМПЕРАТОР в променливи на пролог. Вместо това ще използваме базата знания на пролог.

Да припомним, че вграденият предикат `assert` може да се използва, за да добавим факти към базата знания на пролог. Например след изпълнението на `assert(a(5))` пролог „ще знае“, че `a(5)` е вярно. Предикатът `retract` пък прави обратното, например след изпълнението на `retract(a(5))` пролог „ще забрави“, че `a(5)` е вярно.

За да запомним, че стойността на променливата `x` е 5, ще добавим към базата знания на пролог факт от вида

```
стойност_на_променлива(x, 5)
```

За да дадем на променливата `x` нова стойност, например 7, първо трябва да премахнем от базата знания старата стойност, т.е. факта, имащ вида `стойност_на_променлива(x, _)` и едва след това да добавим новия факт `стойност_на_променлива(x, 7)`.

Следва пълната реализация на ИМПЕРАТОР:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% © 2015 Антон Зиновиев
%
% Тази програма може бъде използвана без ограничения от
% всеки, който е получил нейно копие, в това число да бъде
% разпространявана, изменяна, лицензирана и прелицензирана
% при условие, че тази бележка за авторските права не се
% трие.  ПРОГРАМАТА СЕ ПРЕДОСТАВЯ БЕЗ КАКВИТО И ДА Е
% ГАРАНЦИИ, ЧЕ СТАВА ЗА НЕЩО СМИСЛЕНО И НЯМА ДА НАВРЕДИ.
% Без изрично разрешение, имената на носителите на
% авторските права не могат да бъдат използвани за
% популяризиране на продукти, базирани на тази програма.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op(1150, fx, [император]).
:- op(990, xfx, [':=', '<==']).
:- op(990, fx, [пролог]).

:- op(994, fx, [while]).
:- op(995, xfx, [loop]).

:- op(994, fx, [for]).
:- op(993, xfx, [in]).
:- op(992, xfx, ['..']).

:- op(995, fx, [if]).
:- op(993, xfx, [then]).
:- op(994, xfx, [else]).

:- dynamic стойност_на_променлива/2.

%% стойност(Израз, Стойност) -- пресмята в Стойност
%%            стойността на Израз
стойност(Пром, Стойност) :-
    atom(Пром),
    стойност_на_променлива(Пром, Стойност).
стойност(Число, Число) :-
    number(Число).
стойност(A + B, Z) :-
    стойност(A, X), стойност(B, Y),
    Z is X + Y.
```

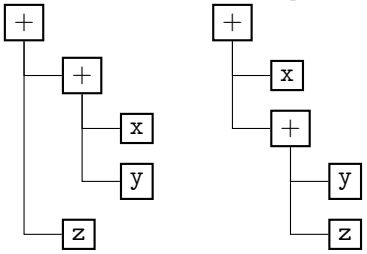
```
стойност(A - B, Z) :-
    стойност(A, X), стойност(B, Y),
    Z is X - Y.
стойност(A * B, Z) :-
    стойност(A, X), стойност(B, Y),
    Z is X * Y.
стойност(A / B, Z) :-
    стойност(A, X), стойност(B, Y),
    Z is X / Y.
стойност(A = B, Z) :-
    стойност(A, X), стойност(B, Y),
    (X = Y -> Z = 1 ; Z = 0).
стойност(A\= B, Z) :-
    стойност(A, X), стойност(B, Y),
    (X \= Y -> Z = 1 ; Z = 0).
стойност(A < B, Z) :-
    стойност(A, X), стойност(B, Y),
    (X < Y -> Z = 1 ; Z = 0).
стойност(A > B, Z) :-
    стойност(A, X), стойност(B, Y),
    (X > Y -> Z = 1 ; Z = 0).
стойност(A =< B, Z) :-
    стойност(A, X), стойност(B, Y),
    (X =< Y -> Z = 1 ; Z = 0).
стойност(A >= B, Z) :-
    стойност(A, X), стойност(B, Y),
    (X >= Y -> Z = 1 ; Z = 0).

оператор X ; Y :-
    (оператор X), (оператор Y).
оператор V := E :-
    once(стойност(E, EVal)),
    %% премахваме старата стойност на V
    (
        retract(стойност_на_променлива(V, _))
    ;
        true
    ), !,
    %% добавяме новата стойност
    assert(стойност_на_променлива(V, EVal)).
оператор V <== E :-
```

```
    once(стойност(E, V)).
император пролог Цел :-
    call(Цел).
император if E then X :-
    (   стойност(E, EVal) , EVal \= 0
    ->  (император X)
    ;   true  ).
император if E then X else Y :-
    (   стойност(E, EVal), EVal \= 0
    ->  (император X)
    ;   (император Y)  ).
император while E loop X :-
    (   стойност(E, EVal), EVal \= 0
    ->  (император X), !, (император while E loop X)
    ;   true ).
император for I in A..B loop X :-
    император
    I := A;
    while I =< B loop (
      X;
      I := I + 1
    ).
```

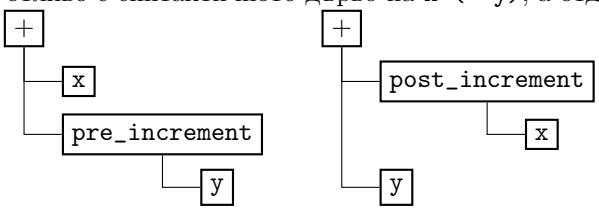
Приложение B

Решения на задачите

Задача 1. Отляво е синтактичното дърво на `(x+y)+z`, `x+y+z`, `((x+y))+z` и `(x)+y+z`, а отдясно — синтактичното дърво на `x+(y+z)`:



Задача 2. Тъй като на си има префикснен `++` и постфиксен `++`, за префиксния ще използваме етикет `pre_increment`, а за постфиксния — `post_increment`. В такъв случай отляво е синтактичното дърво на `x+(++y)`, а отдясно — на `(x++)+y`:



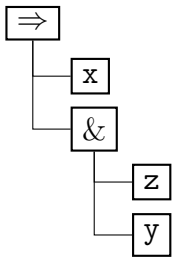
Компилаторите на си ще интерпретират `x+++y` като `(x++)+y`,[**] но не знам дали това е произволно решение или е предписание на стандарта на си.

Задача 3. Индуктивният принцип казва следното:

Нека `p` е свойство, за което е вярно, че:

1. свойството `p` е вярно за 5;
2. свойството `p` е вярно за 8;

[**]И също `x+++++y` като `((x++)++)+y`.



Фиг. 18. Синтактично дърво за $\mathtt{x} \Rightarrow \mathtt{z} \,\&\, \mathtt{y}$

3. ако свойството $\mathfrak{p}$ е вярно за буртантите n и m, то то ще бъде вярно и за nm^2.

Тогава свойството $\mathfrak{p}$ ще бъде вярно за всички буртанти.

Ще използваме този индуктивен принцип в случая, когато $\mathfrak{p}$ е свойството, че числото събрано с 1 се дели на 3 без остатък. За целта трябва да докажем, че са верни трите изисквания в индуктивния принцип за буртанти.

1. Очевидно 5 събрано с 1 се дели на 3 без остатък.
2. Очевидно 8 събрано с 1 се дели на 3 без остатък.
3. Да допуснем, че n и m са такива буртанти, че $n+1$ и $m+1$ се делят на 3 без остатък.* Нека $n+1 = 3k$ и $m+1 = 3l$. Тогава

$$\begin{aligned} nm^2 + 1 &= ((n+1)-1)((m+1)-1)^2 + 1 \\ &= (3k-1)(3l-1)^2 \\ &= 3k(3l-1)^2 - (3l-1)^2 + 1 \\ &= 3k(3l-1)^2 - 9l^2 + 6l - 1 + 1 \\ &= 3(k(3l-1)^2 - 3l^2 + 2l) \end{aligned}$$

Следователно $nm^2 + 1$ се дели на три без остатък.

Доказахме, че трите условия от принципа за индукция за буртанти са изпълнени, от където следва, че всички буртанти събрани с единица се делят на 3 без остатък.

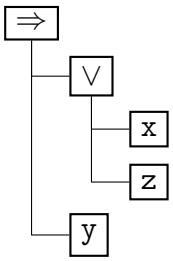
Задача 4. Тъй като изразът $(\mathtt{x} \Leftrightarrow \mathtt{x})$ е съкратен запис за формулата $((\mathtt{x} \Rightarrow \mathtt{x}) \& (\mathtt{x} \Rightarrow \mathtt{x}))$, то значи тази формула съдържа 13 символа.

Задача 5. Първата формула е $(\mathtt{x} \Rightarrow (\mathtt{z} \,\&\, \mathtt{y}))$, а синтактичното ѝ дърво е изобразено във фиг. 18. Втората формула е $((\mathtt{x} \vee \mathtt{z}) \Rightarrow \mathtt{y})$, а синтактичното ѝ дърво е изобразено във фиг. 19. Третата формула е $(\neg(\neg\mathtt{x} \Rightarrow (\neg(\mathtt{y} \,\&\, (\mathtt{z} \vee \mathtt{x})) \,\&\, \neg\mathtt{z})))$, а синтактичното ѝ дърво е изобразено във фиг. 20.

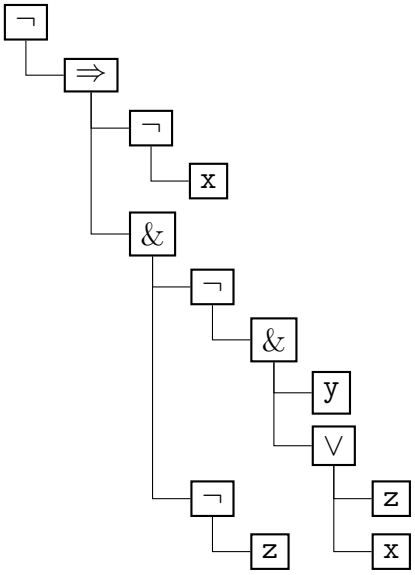
Задача 6. Първото разсъждение по първо правило, второто по второ, третото по трето, четвъртото по четвърто и петото по пето.

Задача 7. $\mathtt{f}$ е двуместен функционален символ, $\mathtt{g}$ и $\mathtt{h}$ са едноместни функционални символи, $\mathtt{c}$ и $\mathtt{a}$ са символи за константи, а $\mathtt{x}$ и $\mathtt{y}$ са променливи.

*Това че $n+1$ и $m+1$ се делят на 3 без остатък е индукционното предположение.



Фиг. 19. Синтактично дърво за $\mathtt{x} \vee \mathtt{z} \Rightarrow \mathtt{y}$



Фиг. 20. Синтактично дърво за $\neg(\neg\mathtt{x} \Rightarrow \neg(\mathtt{y} \,\&\, (\mathtt{z} \vee \mathtt{x})) \,\&\, \neg\mathtt{z})$

Задача 8. $\mathtt{f}(\mathtt{c})$ е терм, $\mathtt{c}(\mathtt{f})$ не е терм, защото константата $\mathtt{c}$ не може да има аргументи, а функционалният символ $\mathtt{f}$ трябва да има аргументи, $\mathtt{c}$ е терм, $\mathtt{f}$ не е терм, защото функционалният символ $\mathtt{f}$ трябва да има поне един аргумент, $\mathtt{x}(\mathtt{c})$ не е терм, защото променливите не могат да имат аргументи, $\mathtt{f}(\mathtt{f}(\mathtt{x}))$ е терм, $\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{c})))$ е терм, $\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{c},\mathtt{c})))$ не е терм, защото $\mathtt{f}$ не може да бъде едновременно едноместен и двуместен, $\mathtt{g}(\mathtt{g}(\mathtt{x},\mathtt{x}),\mathtt{g}(\mathtt{x},\mathtt{y}))$ е терм.

Задача 9. $\mathtt{c}$ е символ за константа и значи терм, а не атомарна формула, $\mathtt{f}(\mathtt{c})$ е терм без променливи, а не атомарна формула, $\mathtt{p}(\mathtt{c})$ и $\mathtt{p}(\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{c}))))$ са атомарни формули без променливи, $\mathtt{p}(\mathtt{f}(\mathtt{x}))$ е атомарна формула, която съдържа променливи, $\mathtt{f}(\mathtt{p}(\mathtt{x}))$ и $\mathtt{p}(\mathtt{p}(\mathtt{x}))$ не са нито термове, нито атомарни формули, $\mathtt{f}(\mathtt{f}(\mathtt{x}))$ не е атомарна формула, а терм, който съдържа променливи.

Задача 10. Тъй като $\mathtt{c}$ е символ за константа, от правило 2.4.4 б) получаваме, че $\mathtt{c}$ е $\mathfrak{sig}$-терм.

Тъй като $\mathtt{x}$ е променлива, от правило 2.4.4 а) получаваме, че $\mathtt{x}$ е $\mathfrak{sig}$-терм.

Тъй като вече сме доказали, че $\mathtt{c}$ и $\mathtt{x}$ са $\mathfrak{sig}$-термове, а $\mathtt{f}$ е двуместен функционален символ от $\mathfrak{sig}$, от правило 2.4.4 в) получаваме, че $\mathtt{f}(\mathtt{c},\mathtt{x})$ е $\mathfrak{sig}$-терм.

Тъй като вече сме доказали, че $\mathtt{c}$ и $\mathtt{f}(\mathtt{c},\mathtt{x})$ са термове, а $\mathtt{f}$ е двуместен функционален символ, от правило 2.4.4 в) получаваме, че $\mathtt{f}(\mathtt{c},\mathtt{f}(\mathtt{c},\mathtt{x}))$ е терм.

Задача 11. Термът $\mathtt{f}(\mathtt{c})$ съдържа скоби, но не съдържа запетаи. Няма термове, които съдържат запетаи, но не съдържат скоби. Във всички термове има равен брой леви и десни скоби.

Задача 12. Например $\mathtt{f}(\mathtt{c},\mathtt{c},\mathtt{c},\mathtt{c},\mathtt{c},\mathtt{c},\mathtt{c})$ и $\mathtt{g}(\mathtt{g}(\mathtt{h}(\mathtt{c},\mathtt{c},\mathtt{c})))$. Във втория от тези два терма може ли да заменим $\mathtt{h}$ с $\mathtt{g}$?

Задача 13. Ако заменим константата с $\mathtt{c}$, едноместния функционален символ с $\mathtt{f}$ и двуместния с $\mathtt{g}$, то тези изрази ще добият следния вид:

$$\boxed{\mathtt{c}}\ \boxed{\mathtt{f}(\mathtt{c})}\ \boxed{\mathtt{f}(\mathtt{f}(\mathtt{c}))}\ \boxed{\mathtt{g}(\mathtt{c},\mathtt{c})}\ \boxed{\mathtt{g}(\mathtt{f}(\mathtt{c}),\mathtt{g}(\mathtt{c},\mathtt{c}))}$$

Задача 14. Термът $+(\mathtt{b},*(\mathtt{x},\mathtt{a}))$ съдържа 4 скоби — две леви и две десни.

Задача 15. С индукция по построението на терма ще докажем, че всеки терм съдържа поне една променлива.

а) Ако $\mathtt{x}$ е променлива, то очевидно $\mathtt{x}$ съдържа променлива.

б) Символи за константи няма.

в) Нека $\mathtt{f}$ е n-местен функционален символ и $\alpha_1,\alpha_2,\ldots,\alpha_n$ са термове, всеки от които съдържа поне една променлива.[*] В такъв случай очевидно и термът $\mathtt{f}(\alpha_1,\alpha_2,\ldots,\alpha_n)$ ще съдържа поне една променлива (тук неявно използваме, че $n \geqq 1$).

Задача 16. а) Ако $\mathtt{x}$ е променлива, то $\mathtt{x}$ съдържа равен брой запетаи и десни скоби (нула), защото променливите не са запазени символи, а запетаята и дясната скоба са.

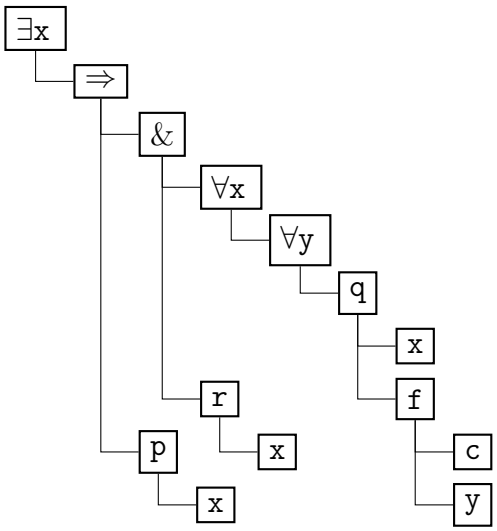
б) Ако $\mathtt{c}$ е символ за константа, то $\mathtt{c}$ съдържа равен брой запетаи и десни скоби (нула), защото символите за константи не са запазени символи, а запетаята и дясната скоба са.

в) Нека $\mathtt{f}$ е функционален символ (по услови той трябва да бъде двуместен). Да допуснем, че α_1 и α_2 са термове с равен брой запетаи и десни скоби.[**] Нека k_i е броят на запетаите в α_i (същото число е равно и на броя на десните скоби). Тъй като $\mathtt{f}$ не е запазен символ, то $\mathtt{f}$ не запетая или скоба. Следователно броят на запетаите в терма $\mathtt{f}(\alpha_1,\alpha_2)$ е $1+k_1+k_2$ — една запетая между α_1 и α_2 плюс запетаите в α_1 и α_2. Броят на десните скоби е същият — затворената скоба в края на терма плюс скобите в α_1 и α_2.

Задача 17. Вж. фиг. 21.

[*] Това е индукционното предположение.

[**] Това е индукционното предположение.



Фиг. 21. Синтактично дърво за формулата .
$\exists \mathtt{x}\,(\forall \mathtt{x}\,\forall \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}))$

Задача 18. В следващия списък са подчертани подформулите:

$$\underline{\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))}$$
$$\forall \mathtt{x}\,\underline{(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))}$$
$$\forall \mathtt{x}\,(\underline{\mathtt{p}(\mathtt{x})} \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \underline{\exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y}))} \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \underline{\exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y}))} \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\underline{\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y}))} \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \underline{\mathtt{r}(\mathtt{x})})$$

В следващия списък са подчертани термове, които също са са поддървета на синтактичното дърво от фигура 11, но не са подформули, защото са термове:

$$\forall \mathtt{x}\,(\mathtt{p}(\underline{\mathtt{x}}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\underline{\mathtt{x}}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \underline{\mathtt{f}(\mathtt{c}, \mathtt{y})}) \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\underline{\mathtt{c}}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \underline{\mathtt{y}})) \,\&\, \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\underline{\mathtt{x}}))$$

Задача 19. Тук е подчертана областта на действие на първия квантор:

$$\underline{\forall \mathtt{x}\,(\forall \mathtt{x}\,\exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{y}))}$$

Тук променливите, управлявани от първия квантор са подчертани:

$$\forall \underline{\mathtt{x}}\,(\forall \mathtt{x}\,\exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\underline{\mathtt{x}}) \Rightarrow \mathtt{p}(\underline{\mathtt{x}}, \mathtt{y}))$$

Тук е подчертана областта на действие на втория квантор:

$$\forall \mathtt{x}\,(\underline{\forall \mathtt{x}\,\exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))}\,\&\,\mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))$$

Тук променливите, управлявани от втория квантор са подчертани:

$$\forall \mathtt{x}\,(\forall \underline{\mathtt{x}}\,\exists \mathtt{y}\,\mathtt{q}(\underline{\mathtt{x}},\mathtt{f}(\mathtt{c},\mathtt{y}))\,\&\,\mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))$$

Тук е подчертана областта на действие на третия квантор:

$$\forall \mathtt{x}\,(\forall \mathtt{x}\,\underline{\exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))}\,\&\,\mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))$$

Тук променливите, управлявани от третия квантор са подчертани:

$$\forall \mathtt{x}\,(\forall \mathtt{x}\,\exists \underline{\mathtt{y}}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\underline{\mathtt{y}}))\,\&\,\mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))$$

Последната променлива $\mathtt{y}$ е единствената свободна променлива, а всички други променливи са свързани. Множеството от свободните променливи на тази формула е $\{\mathtt{y}\}$.

Задача 20. **(а)** Да. Формулите са с една и съща дължина и символите, които не са променливи са едни и същи. Всички променливи са свързани и ако в едната формула на дадена позиция има свързана променлива, то и в другата има свързана променлива. Кванторът на променливата $\mathtt{x}$ от $\mathtt{p}(\mathtt{x},\mathtt{y})$ в първата формула е първият квантор, а кванторът на променливата $\mathtt{x}$ от $\mathtt{p}(\mathtt{x},\mathtt{y})$ във втората формула също е първият квантор. Кванторът на променливата $\mathtt{y}$ от $\mathtt{p}(\mathtt{x},\mathtt{y})$ в първата формула е последният квантор, а кванторът на променливата $\mathtt{y}$ от $\mathtt{p}(\mathtt{x},\mathtt{y})$ във втората формула също е последният квантор.

(б) Да. Формулите са с една и съща дължина и символите, които не са променливи са едни и същи. Всички променливи са свързани и ако в едната формула на дадена позиция има свързана променлива, то и в другата има свързана променлива. Кванторът на променливата $\mathtt{x}$ от $\mathtt{p}(\mathtt{x},\mathtt{y})$ в първата формула е първият квантор, а кванторът на променливата $\mathtt{y}$ от $\mathtt{p}(\mathtt{y},\mathtt{x})$ във втората формула също е първият квантор. Кванторът на променливата $\mathtt{y}$ от $\mathtt{p}(\mathtt{x},\mathtt{y})$ в първата формула е последният квантор, а кванторът на променливата $\mathtt{x}$ от $\mathtt{p}(\mathtt{y},\mathtt{x})$ във втората формула също е последният квантор.

(в) Не. В първата от формулите първият аргумент на $\mathtt{p}$ се свързва от първия квантор, а във втората формула същият аргумент се свързва от втория квантор.

Задача 21. Една такава формула е например следната:

$$\forall \mathtt{x}'\forall \mathtt{y}'(\exists \mathtt{x}''\mathtt{p}(\mathtt{x}'',\mathtt{y}')\,\&\,\mathtt{q}(\mathtt{x}',\mathtt{z}) \Rightarrow \exists \mathtt{z}'\mathtt{p}(\mathtt{y}',\mathtt{z}'))$$

Задача 22. Нека $\mathtt{z}$ е свободна променлива на φ и $\mathtt{z} \neq \mathtt{x}$. Да разгледаме някое свободно срещане на $\mathtt{z}$ в φ и съответното срещане на $\mathtt{z}$ в ψ. Ако допуснем, че срещането на $\mathtt{z}$ в ψ не е свободно, то значи това срещане в ψ ще попада в областта на действие на някой квантор $\forall \mathtt{z}$ или $\exists \mathtt{z}$. При преименуването обаче по никакъв начин не влияем на променливата $\mathtt{z}$, следователно областта на действие на същия този квантор в φ ще бъде същата и значи ще включва и срещането на $\mathtt{z}$, за което допуснахме, че е свободно. Това е противоречие.

Задача 23. Проверяваме едно по едно условията от дефиницията за конгруентност. Дължината на φ и φ' е една и съща, защото ψ и ψ' са с една и съща дължина. Символите, които не са променливи, в φ и φ' са едни и същи, защото в ψ и ψ' те са едни и същи.

Ако $\mathtt{x}$ е свободна променлива в φ и това срещане на $\mathtt{x}$ не е в подформулата ψ, то тогава на същата позиция в φ' стои същата променлива и тя е свободна. Ако това срещане на $\mathtt{x}$ е в ψ, то значи то е свободно не само в φ, но и в ψ (защото ако си имаше квантор в ψ, то същият квантор би управлявал променливата е в φ). Щом това срещане е свободно и в ψ и $\psi \equiv \psi'$, то значи на същата позиция в ψ' стои същата променлива и тя е свободна. Тя ще бъде свободна и в φ', защото ако си имаше квантор в φ', то този квантор би бил извън ψ' и значи същият квантор щеше да го има е в φ и променливата $\mathtt{x}$ не би била свободна в φ.

Нека $\mathtt{x}$ е свързана променлива в φ и управляващият квантор не е от ψ. Ако това срещане на $\mathtt{x}$ не е от ψ, тогава на същата позиция в φ' стои същата променлива и управляващият квантор е отново същият. Ако пък това срещане на $\mathtt{x}$ е от ψ, то значи то е свободно в ψ (защото управляващият квантор е извън ψ). Тогава от $\psi \equiv \psi'$ получаваме, че на същата позиция в ψ' стои същата променлива и тя е свободна. Следователно управляващият квантор на $\mathtt{x}$ в φ ще бъде управляващ на $\mathtt{x}$ и в φ'.

Нека $\mathtt{x}$ е свързана променлива в φ и управляващият квантор е от ψ. Тогава $\mathtt{x}$ ще бъде свързана и в ψ. От $\psi \equiv \psi'$ получаваме, че на съответната позиция в ψ' ще стои свързана променлива, чийто управляващ квантор е на същата позиция. Същият квантор ще бъде управляващ и в φ'.

Задача 24. Непосредствено проверяваме условията от дефиницията за конгруентност.

Задача 25.

$$\begin{aligned}
\mathtt{f(x,y)[x,y \coloneqq y,y]} &= \mathtt{f(y,y)} \\
\mathtt{f(x,y)[x,y \coloneqq g(x),x]} &= \mathtt{f(g(x),x)} \\
\mathtt{f(g(x),y)[x,y \coloneqq g(x),x]} &= \mathtt{f(g(g(x)),x)} \\
\mathtt{f(x,g(y))[x,y \coloneqq g(x),x]} &= \mathtt{f(g(x),g(x))}
\end{aligned}$$

Задача 26. Тъй като субституцията не променя променливата $\mathtt{y}$, ще обясним какво се случва само с променливите $\mathtt{x}$.

(а) Първата променлива $\mathtt{x}$ е свързана и затова не я пипаме. Втората променлива $\mathtt{x}$ можем да заменим без проблеми. Отговор: $\forall \mathtt{x}\, \mathtt{p(x,y)} \vee \mathtt{q(f(y))}$.

(б) Ако просто заменим променливата $\mathtt{x}$ с $\mathtt{f(y)}$ ще получим грешка от втори вид, защото променливата $\mathtt{y}$ в $\mathtt{f(y)}$ ще се свърже с квантора $\forall \mathtt{y}$. Затова най-напред трябва да преименуваме свързаната променлива $\mathtt{y}$ например на $\mathtt{z}$ и след това да заменим $\mathtt{x}$ с $\mathtt{f(y)}$. Отговор: $\forall \mathtt{z}\, \mathtt{p(f(y),z)} \vee \mathtt{q(y)}$. Забележете, че свободната променлива $\mathtt{y}$ не се преименува!

Задача 27. При прилагане на s към първата формула получаваме

$$\mathtt{p(f(x_1 + x_2, z), (t + x_3) + x_1)} \vee \mathtt{f(x_1 + x_2, z) + (y + x_3) = f(x_1 + f(x_2, t), f(y, y))}$$

Тъй като останалите формули съдържат квантори, то трябва да проверим дали няма квантори с опасни променливи.

Свободни променливи на втората формула са: $\mathtt{t}, \mathtt{x}_1$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{t}), s(\mathtt{x}_1)$, т. е. променливите $\mathtt{x}_1, \mathtt{x}_2, \mathtt{t}, \mathtt{z}$. И двата квантора са с променливи, които не са измежду опасните, следователно не се налага да преименуваме. Както обикновено, когато прилагаме субституцията, трябва да заменяме само свободните променливи. Получаваме

$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{x}_1 + \mathtt{f}(\mathtt{x}_2, \mathtt{t})) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{f}(\mathtt{x}_1 + \mathtt{x}_2, \mathtt{z}) + (\mathtt{x}_1 + \mathtt{f}(\mathtt{x}_2, \mathtt{t}))))$$

Свободни променливи на третата формула са: $\mathtt{z}, \mathtt{x}_1$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{z}), s(\mathtt{x}_1)$, т. е. променливите $\mathtt{t}, \mathtt{x}_3, \mathtt{x}_1, \mathtt{x}_2, \mathtt{z}$. Кванторът $\forall \mathtt{x}_3$ е с опасна променлива и трябва да го преименуваме. Например ако преименуваме неговата променлива $\mathtt{x}_3$ на $\mathtt{x}_{352}$ ще получим

$$\forall \mathtt{x}_{352}(\mathtt{p}(\mathtt{x}_{352}, \mathtt{z}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_{352}) = \mathtt{x}_1 + \mathtt{z}))$$

Към тази формула вече може да прилагаме субституцията s (заменяме само свободните променливи). Получаваме

$$\forall \mathtt{x}_{352}(\mathtt{p}(\mathtt{x}_{352}, (\mathtt{t} + \mathtt{x}_3) + \mathtt{x}_1) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_{352}) = \mathtt{f}(\mathtt{x}_1 + \mathtt{x}_2, \mathtt{z}) + ((\mathtt{t} + \mathtt{x}_3) + \mathtt{x}_1)))$$

Свободни променливи на четвъртата формула са: $\mathtt{z}, \mathtt{x}_2$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{z}), s(\mathtt{x}_2)$, т. е. променливите $\mathtt{t}, \mathtt{x}_3, \mathtt{x}_1, \mathtt{y}$. И двата квантора се оказват с опасна променлива и трябва да ги преименуваме. Например ако преименуваме променлива $\mathtt{x}_3$ на $\mathtt{x}_{352}$ и $\mathtt{y}$ на $\mathtt{y}_{178}$ ще получим

$$\forall \mathtt{x}_{352}(\mathtt{p}(\mathtt{x}_{352}, \mathtt{z}) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_{352}) = \mathtt{x}_2 + \mathtt{z}))$$

Към тази формула вече може да прилагаме субституцията s (заменяме само свободните променливи). Получаваме

$$\forall \mathtt{x}_{352}(\mathtt{p}(\mathtt{x}_{352}, (\mathtt{t} + \mathtt{x}_3) + \mathtt{x}_1) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_{352}) = (\mathtt{y} + \mathtt{x}_3) + ((\mathtt{t} + \mathtt{x}_3) + \mathtt{x}_1)))$$

Задача 28. Най-напред да разгледаме случая когато в φ няма квантори с опасни променливи при субституцията s. Тъй като субституцията замества свободни променливи с термове, свързаните променливи в φ си остават свързани и значи не са измежду свободните променливи на φs. От друга страна променливите в термовите, с които свободните променливи в φ са заместени, не се хващат от никой квантор (защото иначе би имало квантори с опасни променливи). Това доказва исканото.

За да видим общия случай, нека $\varphi \equiv \varphi'$ и в φ' няма квантори с опасни променливи при субституция s. Тъй като конгруентните формули имат едни и същи свободни променливи и от $\varphi \equiv \varphi'$ следва $\varphi s \equiv \varphi' s$, то исканото следва от вече доказаното.

Задача 29. Ще проверим $(\varphi \,\&\, \psi)s \equiv \varphi s \,\&\, \psi s$, останалите условия се доказват аналогично. Нека φ' е формула, конгруентна с φ, в която няма квантори с опасни променливи при субституция s и аналогично ψ' за ψ. Тогава $\varphi' \,\&\, \psi' \equiv \varphi \,\&\, \psi$ и в $\varphi' \,\&\, \psi'$ няма квантори с опасни променливи при субституция s. Тъй като s се прилага към φ', ψ' и $\varphi' \,\&\, \psi'$ просто заместваики свободните променливи с термове, то $(\varphi' \,\&\, \psi')s \equiv \varphi' s \,\&\, \psi' s$. От тук следва исканото.

Задача 30. ($\Longrightarrow$) Променливата $\mathtt{y}$ не е свободна променлива на $\forall \mathtt{x}\,\varphi$, защото тя не е свободна променлива на $\forall \mathtt{y}\,\psi$, а конгруентните формули имат едни и същи променливи.

Нека $\mathtt{z}$ е променлива, която не се среща никъде в $\forall \mathtt{x}\,\varphi$ и $\forall \mathtt{y}\,\psi$. Тъй като във φ няма опасни променливи при субституцията $[\mathtt{x} := \mathtt{z}]$, поглеждайки дефиницията на конгруентност, забелязваме, че $\forall \mathtt{x}\,\varphi \equiv \forall \mathtt{z}\,(\varphi[\mathtt{x} := \mathtt{z}])$. Аналогично $\forall \mathtt{y}\,\psi \equiv \forall \mathtt{z}\,(\psi[\mathtt{y} := \mathtt{z}])$. Следователно $\forall \mathtt{z}\,(\varphi[\mathtt{x} := \mathtt{z}]) \equiv \forall \mathtt{z}\,(\psi[\mathtt{y} := \mathtt{z}])$ и значи от твърдение 2.6.13 получаваме $\varphi[\mathtt{x} := \mathtt{z}] \equiv \psi[\mathtt{y} := \mathtt{z}]$ и значи $(\varphi[\mathtt{x} := \mathtt{z}])[\mathtt{z} := \mathtt{y}] \equiv (\psi[\mathtt{y} := \mathtt{z}])[\mathtt{z} := \mathtt{y}]$. От тук, използвайки 2.7.17, получаваме $\varphi[\mathtt{x}, \mathtt{z} := \mathtt{y}, \mathtt{y}] \equiv \psi[\mathtt{y}, \mathtt{z} := \mathtt{y}, \mathtt{y}]$. Тъй като $\mathtt{z}$ не се среща нито в φ, нито в ψ, то значи е без значение с какво $\mathtt{z}$ се замества от тези субституции и значи $\varphi[\mathtt{x} := \mathtt{y}] \equiv \psi$.

($\Longleftarrow$) Нека $\mathtt{z}$ е променлива, която не се среща никъде в $\forall \mathtt{x}\,\varphi$ и $\forall \mathtt{y}\,\psi$. Тъй като във φ няма опасни променливи при субституцията $[\mathtt{x} := \mathtt{z}]$, поглеждайки дефиницията на конгруентност, забелязваме, че $\forall \mathtt{x}\,\varphi \equiv \forall \mathtt{z}\,(\varphi[\mathtt{x} := \mathtt{z}])$. Аналогично $\forall \mathtt{y}\,\psi \equiv \forall \mathtt{z}\,(\psi[\mathtt{y} := \mathtt{z}]) \equiv \forall \mathtt{z}\,((\varphi[\mathtt{x} := \mathtt{y}])[\mathtt{y} := \mathtt{z}]) \equiv \forall \mathtt{z}\,(\varphi[\mathtt{x}, \mathtt{y} := \mathtt{z}, \mathtt{z}])$. Когато $\mathtt{x} = \mathtt{y}$, последното е конгруентно на $\forall \mathtt{z}\,(\varphi[\mathtt{x} := \mathtt{z}])$, което вече видяхме, че е конгруентно на $\forall \mathtt{x}\,\varphi$. Когато пък $\mathtt{x} \neq \mathtt{y}$, щом $\mathtt{y}$, не е свободна в $\forall \mathtt{x}\,\varphi$, то значи тя не е свободна и в φ и значи отново имаме $\forall \mathtt{z}\,(\varphi[\mathtt{x}, \mathtt{y} := \mathtt{z}, \mathtt{z}]) \equiv \forall \mathtt{z}\,(\varphi[\mathtt{x} := \mathtt{z}]) \equiv \forall \mathtt{x}\,\varphi$.

Задача 31. Непосредствено от дефиницията се проверява, че $\forall \mathtt{x}\,\varphi \equiv \forall \mathtt{y}\,\psi$. Затова исканото следва от задача 30.

Задача 32. Съгласно задача 30 $\varphi' \equiv \varphi[\mathtt{x} := \mathtt{x}']$ и $\psi' \equiv \psi[\mathtt{x} := \mathtt{x}']$. Пак от същата задача следва, че е достатъчно да докажем, че $\varphi' \,\&\, \psi' \equiv (\varphi \,\&\, \psi)[\mathtt{x} := \mathtt{x}']$. Последното е вярно, защото съгласно задача 29 $(\varphi \,\&\, \psi)[\mathtt{x} := \mathtt{x}'] \equiv \varphi[\mathtt{x} := \mathtt{x}'] \,\&\, \psi[\mathtt{x} := \mathtt{x}']$.

Задача 33. (6) За произволни реални числа x, y и z ако $x < y$ и $y < z$, то $x < z$.

(7) За произволни реални числа x, y и z ако $x < y$, то $x + z < y + z$.

(8) Ако $3 < 3$, то всяко реално число е по-малко от себе си.

Задача 34. 1. $\mathtt{p}(\mathtt{x}, \mathtt{y}) \Rightarrow \mathtt{p}(\mathtt{y}, \mathtt{x})$

2. $\mathtt{p}(\mathtt{x}, \mathtt{y})$

Задача 35. Тъй като универсумът на $\mathbf{M}$ е непразно множество, то той съдържа поне един елемент μ. Да дефинираме оценката v така: $v(\mathtt{x}) = \mu$ за всяка променлива $\mathtt{x}$.

Задача 36. Тъй като τ не съдържа променливи, то $v(\mathtt{x}) = v'(\mathtt{x})$ за всяка променлива $\mathtt{x}$, която се среща в τ. Затова исканото следва от твърдение 3.2.5.

Задача 37. Нека термът τ е $\mathtt{f}(\mathtt{g}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{x}_2)$.

Задача 38. Нека термът τ е $\mathtt{f}(\mathtt{g}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{x}_2)$ и формулата φ е $\mathtt{p}(\mathtt{g}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{f}(\mathtt{x}_2, \mathtt{x}_2))$.

Задача 39. Непосредствено от дефиницията следва, че φ не е изпълнима в $\mathbf{M}$ тогава и само тогава, когато φ е неизпълнима в $\mathbf{M}$.

Ако φ е неизпълнима, то не съществува оценка, при която φ е вярна, следователно каквато и оценка да изберем, φ няма да е вярна при тази оценка и значи φ е невярна при всяка оценка, т.е. φ е тъждествено невярна.

Ако φ е тъждествено невярна в $\mathbf{M}$, то φ е невярна при всяка оценка, значи няма как да намерим оценка, при която φ е вярна, следователно φ е неизпълнима.

Задача 40. Непосредствено от дефинициите следва, че φ е неизпълнима тогава и само тогава, когато не е вярно, че съществува структура, в която φ е тъждествено вярна. И също, че φ е изпълнима тогава и само тогава, когато съществува структура, в която φ е тъждествено вярна. Т.е. едното е отрицание на другото.

Формулата $\neg\varphi$ е тъждествено вярна тогава и само тогава, когато $\neg\varphi$ е вярна за произволни структура и оценка. Това пък е така тогава и само тогава, когато φ е невярна за произволни структура и оценка. Т.е. когато φ е тъждествено невярна.

Задача 41. Да. Например атомарната формула $\mathtt{x} = \mathtt{c}$ е изпълнима във всяка структура с равенство, защото е вярна когато оценката дава на $\mathtt{x}$ същата стойност, каквато структурата дава на $\mathtt{c}$. Но от друга страна тази формула няма да бъде тъждествено вярна, ако в универсума на структурата се съдържат поне два елемента, защото тогава ще съществуват оценки, при които стойността на $\mathtt{x}$ се различава от стойността на $\mathtt{c}$.

Задача 42. Ако φ е тъждествено вярна, то φ е вярна при всяка оценка. Нека v е произволно избрана оценка. Тогава φ е вярна при оценка v, значи φ е изпълнима в структурата и следователно не е неизпълнима в структурата.

В това доказателство използваме, че универсумът на структурата е непразно множество, по следния начин: за да можем да изберем произволна оценка v, е нужно да сме сигурни, че съществува поне една оценка. Ако ни е даден поне един елемент от универсума, тогава със сигурност ще съществува поне една оценка — да вземем например оценката, която на всички променливи дава като стойност дадения елемент от универсума. Ако обаче универсумът бе празното множество, то не би съществувала нито една оценка, защото за произволна променлива $\mathtt{x}$, $v(\mathtt{x})$ трябва да бъде елемент на универсума.

Задача 43. Задачата може да се реши лесно, използвайки свойствата на хомоморфизмите, които ще докажем по-нататък, но всъщност тя има много и най-различни решения. Ето едно (упътване):

Нека $\mathbf{M}$ е модел за Γ. За всяко естествено число m ще дефинираме различни по между си структури $\mathbf{K}_m$, които са модели за Γ. Универсум на $\mathbf{K}_m$ е $|\mathbf{M}| \times \{m\}$. За всеки символ за константа нека $\mathtt{c}^{\mathbf{K}_m} = \langle \mathtt{c}^{\mathbf{K}}, m\rangle$. За всеки функционален символ нека $\mathtt{f}^{\mathbf{K}_m}(\langle\mu_1, m\rangle, \langle\mu_2, m\rangle, \ldots, \langle\mu_n, m\rangle) = \langle \mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n), m\rangle$. И за всеки предикатен символ нека $\mathtt{p}^{\mathbf{K}_m}(\langle\mu_1, m\rangle, \langle\mu_2, m\rangle, \ldots, \langle\mu_n, m\rangle) = \mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)$.

За произволна оценка v в $\mathbf{K}_m$ да означим с v' оценката в $\mathbf{M}$, такава че за произволна променлива $\mathtt{x}$, $v'(\mathtt{x})$ е равно на първия елемент на наредената двойка $v(\mathtt{x})$.

С индукция по термовете докажете, че за произволен терм τ и оценка v в $\mathbf{K}_m$, ако μ е стойността на τ в $\mathbf{M}$ при оценка v', то $\langle\mu, m\rangle$ е стойността на τ в $\mathbf{K}_m$ при оценка v.

След това докажете още, че за произволна атомарна формула φ и оценка v в $\mathbf{K}_m$, стойността на φ в $\mathbf{M}$ при оценка v' е еквивалентна на стойността на φ в $\mathbf{K}_m$ при оценка v.

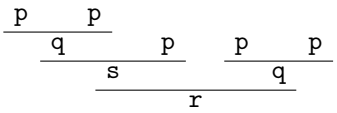
От тук заключете, че ако някоя формула е тъждествено вярна в $\mathbf{M}$, то тя ще бъде тъждествено вярна и в $\mathbf{K}_m$ и обратно.

Задача 44. Разбира се, че е възможно. В твърдение 3.4.9 се говори за изпълнимост и тъждествена вярност в конкретна структура, а не за изпълнимост и тъждествена вярност по принцип. Например формулата $\forall\mathtt{x}\,\forall\mathtt{y}\,(\mathtt{x}.\mathtt{y} = \mathtt{y}.\mathtt{x})$ е изпълнима, защото е вярна във всяка структура, която е абелева група или комутативен пръстен, но не е тъждествено вярна (не е предикатна тавтология), защото не е вярна в структура, която е неабелева група или некомутативен пръстен.

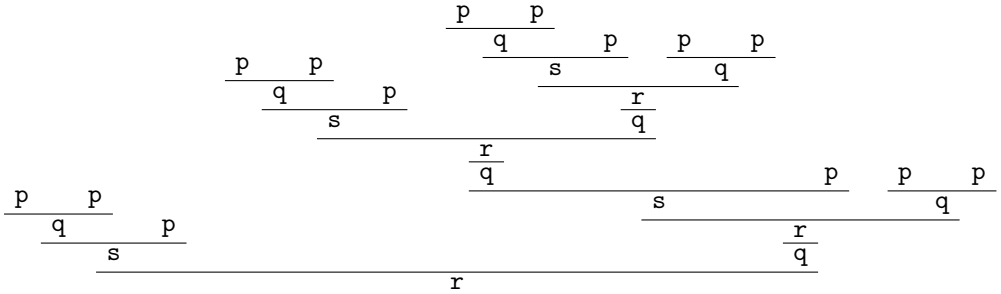
Задача 45.

1. Ако структурата $\mathbf{M}$ е такава, че в нея са тъждествено верни всички елементи на Γ, то в частност и φ ще бъде тъждествено вярно в $\mathbf{M}$.
2. следва от в).
3. Да допуснем, че φ следва от Δ. Да изберем произволна структура, в която са тъждествено верни елементите на Γ. Тъй като всеки от елементите на Δ следва от Γ, то в тази структура ще бъдат тъждествено верни и елементите на Δ. Но φ следва от Δ, значи в тази структура и φ е тъждествено вярно.

Задача 46. Един възможен извод е следният:



Ето друг възможен извод:



Кой е възможно най-големият извод на `r`, който можете да измислите?

Задача 47. (*Упътване.*) Първо да докажем следната лема: ако елементите на X могат да се генерират алгортмично, то елементите на $T_\Gamma(X)$ също могат да се генерират алгоритмично.

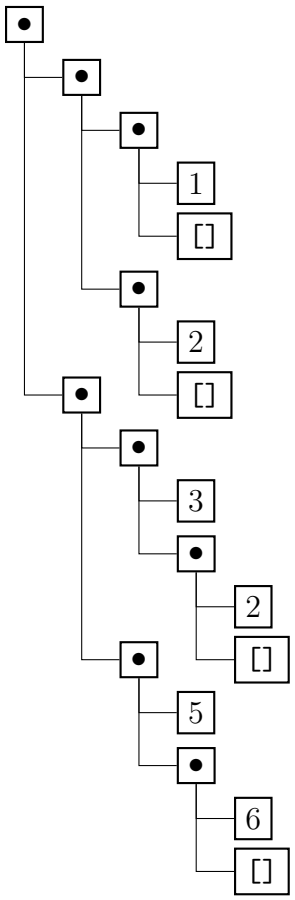
(*Доказателство на лемата.*) Да пуснем като паралелни фонови процеси алгоритмите, генериращи елементите на X и частните случаи на Γ. Във всеки момент от времето ще сме получили краен брой елементи на X и краен брой частни случаи на елементи на Γ. С тези краен брой елементи може да се изведат едностъпково само краен брой атомарни формули (и не е трудно да установим кои са те). Нека с всеки новополучен елемент на X или Γ прегенерираме атомарните формули, които могат да се изведат от тях едностъпково. Елементите на $T_\Gamma(X)$ са всички атомарни формули, които могат да се получат по този начин.

Сега ще използваме доказаната лема. Ясно е, че елементите на $\varnothing$ могат да се генерират алгоритмично. Тогава лемата казва, че елементите на $T_\Gamma(\varnothing)$ могат да се генерират алгоритмично. От това пък пак с лемата получаваме, че и елементите на $T^2_\Gamma(\varnothing)$ могат да се генерират алгоритмично. Да пускаме все повече и повече паралелни фонови процеси генериращи елементите на $T^0_\Gamma(\varnothing), T^1_\Gamma(\varnothing), T^2_\Gamma(\varnothing), T^3_\Gamma(\varnothing), \ldots$. Елементите, които ще генерира някой от тези процеси са елементите на $T^\infty_\Gamma(\varnothing)$.

Задача 48. Бърз преглед на дефиницията на структура 3.1.5 показва, че за да дефинираме коя да е структура (вкл. ербранова структура) трябва да определим следното: кой е универсумът, как се интерпретират символите за константите, как се интерпретират функционалните символи и как се интерпретират предикатните символи.

В случая можем да вземем универсъмът на структурата да бъде множеството от всички термове без променливи. Да забележим, че това множество е непразно, защото сигнатурата съдържа поне един символ за константа и този символ е терм без променливи.

Интерпретацията на символите за константи и функционалните символи на всяка една ербранова структура е еднозначно определена, така че не се налага да я измисляме: $\mathtt{c}^{\mathbf{H}} = \mathtt{c}$ и $\mathtt{f}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Няма никакви ограничения за интерпретацията на предикатните символи. За определеност може да считаме, че $\mathtt{p}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_N)$ е истина за всеки предикатен символ $\mathtt{p}$ и елементи $\tau_1, \tau_2, \ldots, \tau_n$ на универсума на $\mathbf{H}$.



Фиг. 22. Синтактично дърво за `[[[1|[]]|[2|[]]]|[[3|[4|[]]]|[5|[6|[]]]]]` и `•(•(•(1,[]),•(2,[])),•(•(3,•(4,[])),•(5,•(6,[]))))`

Задача 49. Ако означаваме скобите и запетаите, които са символи, с $\texttt{,}$, $\texttt{(}$ и $\texttt{)}$, тогава можем да препишем израза от условието на задачата така:

$$\texttt{f}\big(\texttt{g}^{\mathbf{H}}(v(\texttt{x}),(\texttt{g}\big(\texttt{a}\texttt{,}\texttt{y}\big))v)\big)$$

Стойността на този израз в $\mathbf{H}$ е термът $\texttt{f}(\texttt{g}(\texttt{f}(\texttt{f}(\texttt{c})),\texttt{g}(\texttt{a},\texttt{g}(\texttt{c},\texttt{f}(\texttt{a})))))$. С просто преброяване установяваме, че той съдържа 7 леви скоби и 3 запетаи.

Задача 50. Не. Например дизюнктът $\{\texttt{p}(\texttt{x}),\texttt{p}(\texttt{c})\}$ има частен случай $\{\texttt{p}(\texttt{c})\}$, който се получава при прилагане на субституцията $[\texttt{x} := \texttt{c}]$.

Задача 51. `[[[1],2],[3,4],5,6]`. За синтактичното дърво вж. фиг. 22.

Задача 52. `:- op(800,fy,[not]).`

Задача 53.

```
:- op(800,fy,[не]).
:- op(1000, xfy, [и]).
:- op(1100, xfy, [или]).
не Нещо :- not(call(Нещо)).
Едно и Друго :- call(Едно), call(Друго).
Едно или Друго :- call(Едно); call(Друго).
```

Приложение Г

Задължителни неща

На изпита по логическо програмиране се очаква всички студенти да са научили нещата, означени с ✓. Познаването само на тези неща не гарантира получаването висока оценка. Не е нужно да се помнят условията на задачите, включени в този списък, но трябва да може да се решава всяка задача с подобно условие.

При твърденията, които трябва да се знаят без доказателство, от студентите се очаква да се сещат сами за съответното твърдение, ако то е нужно за решаването на някоя задача. Да разгледаме например следната задача:

> Нека структурата $\mathbf{M}$ е с универсум множеството на естествените числа, а оценките v и w в $\mathbf{M}$ са такива, че $v(\mathtt{x}) = 5$, $v(\mathtt{y}) = 8$, $v(\mathtt{z}) = 3$, $w(\mathtt{x}) = 7$, $w(\mathtt{y}) = 8$, $w(\mathtt{z}) = 9$. Да се докаже, че стойността на формулата
>
> $$\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \Rightarrow \exists \mathtt{z}\, \forall \mathtt{x}\, (\neg \mathtt{q}(\mathtt{z}, \mathtt{x}, \mathtt{y}) \,\&\, \mathtt{p}(\mathtt{x}, \mathtt{y}))$$
>
> в $\mathbf{M}$ при оценка v е еквивалентна на стойността ѝ в $\mathbf{M}$ при оценка w.

От студентите се очаква да могат да дадат приблизително следното решение:

Променливата $\mathtt{y}$ е единствената свободна променлива в тази формула. Тъй като $v(\mathtt{y}) = w(\mathtt{y}) = 8$, то исканата еквивалентност следва от следното твърдение:

Твърдение. *Нека v и v' са оценки в структура $\mathbf{M}$. Ако v и v' съвпадат за всички свободни променливи на формулата φ, то $\mathbf{M} \vDash \varphi[v]$ е еквивалентно на $\mathbf{M} \vDash \varphi[v']$.*

Не е нужно твърденията да могат да се цитират точно по начина, по който са били дадени на лекциите или са формулирани в записките. Разбира се, твърденията, които се цитират, трябва да са верни, а ако има съществени разлики между цитираното твърдение, и твърдението, формулирано в записките, тогава се очаква при поискване формулираното твърдение да може да се докаже.

Приложение Д

Скоростна класация

Ще направим наивно сравнение на скоростта на различни езици за програмиране. За целта ще използваме следната задача. Ще казваме, че едно естествено число е *добро*, ако то има нечетен брой прости делители, които са по-малки от самото число. Задачата е да се преброи колко добри числа има между 1 и 25000.

Задачата ще се решава по възможно най-директния начин. Например, за да проверим дали p е просто, ще търсим потенциални делители между 2 и $p/2$, а не между 2 и $\sqrt{p}$. Освен това за всяко число между 2 и $p/2$ ще проверяваме дали е делител, макар че единственото четно просто число е 2, а когато p е нечетно, няма смисъл да търсим четни делители.

Резултатите са представени в таблица 6. Числото в предпоследния стълб показва колко секунди е работила програмата. Числото в последният стълб показва производителността, като скоростта на програмата на си, компилирана с GCC е приета условно за 100%.

Тестовете са правени на компютър с процесор Intel© Core™ i5-4440 (3,10 GHz) под управлението на Linux Mint.

Следват текстовете на програмите, участвали в теста.

език	*транслатор*	*секунди*	*производ.*
ада	GNATMAKE 14.0.1	0,475	100,7%
фортран	GNU Fortran 13.2.0	0,4772	100,3%
Си	GCC 13.2.0	0,4785	100,0%
Си	Clang 18.1.3	0,7417	64,5%
чепъл	Chapel 2.2.0	0,52	92,0%
ръст	rustc 1.75.0	0,6855	69,8%
джава	OpenJDK 21.0.5	0,6936	69,0%
Си#	Mono C# compiler 6.8.0.105	0,6957	68,8%
суифт	swiftc 6.0.2, компилатор	0,7422	64,5%
суифт	swiftc 6.0.2, интерпретатор	52,62	0,9%
джаваскрипт	Node.js v12.22.9	0,9725	49,2%
го	GCCgo 13.2.0	1,543	31,0%
го	Go 1.18.1	1,589	30,1%
паскал	Free Pascal Compiler 3.2.2	1,575	30,4%
форт	Gforth 0.7.3	2,361	20,3%
мъркюри	Mercury Compiler 22.01.8	2,554	18,7%
пайтън	PyPy 7.3.9 + Python 3.8.13	2,782	17,2%
пайтън	Python 3.10.12	13,19	3,6%
окамъл	OCaml 4.13.1, компилатор	3,246	14,7%
окамъл	OCaml 4.13.1, интерпретатор	6,725	7,1%
скийм	Chez Scheme 9.5.4	3,64	13,1%
скийм	GNU Guile 3.0.7	9,681	4,9%
скийм	CHICKEN Scheme 5.2.0	26,28	1,8%
пролог	GNU Prolog 1.4.5, компилатор	4,074	11,7%
пролог	SICStus Prolog 4.9.0	4,967	9,6%
пролог	Ciao 1.24.0-m1 (ciaoc)	10,22	4,7%
пролог	GNU Prolog 1.4.5, интерпретатор	10,46	4,6%
пролог	B-Prolog 8.1	15,09	3,2%
пролог	YAP Prolog 6.5.0	29,14	1,6%
пролог (строубери)	Strawberry Prolog 6.1	31,55	1,5%
пролог	SWI-Prolog 8.4.2	32,18	1,5%
Пи Ейч Пи	PHP 8.1.2	6,332	7,6%
пърл	Perl 5.34.0	8,067	5,9%
хаскел	GHC 8.8.4, компилатор	10,49	4,6%
хаскел	GHCi 8.8.4, интерпретатор	69,51	0,7%
R	R 4.1.2	11,56	4,1%
руби	Ruby 3.0.2p107	13,52	3,5%
бейсик	Bywater BASIC 2.20 patch level 2	1352,5	0,0%

Таблица 6. Скоростна класация на езиците за програмиране

Програма на ада:

```
with Ada.Integer_Text_IO;

procedure Ada_Benchmark is

   function Is_Prime(N: Integer) return Boolean is
      (for all J in 2 .. N/2 => N rem J /= 0);

   function Is_Good(N: Integer) return Boolean is
      Count : Integer := 0;
   begin
      for J in Integer range 2 .. N/2 loop
         if N rem J = 0 and then Is_Prime(J) then
            Count := Count + 1;
         end if;
      end loop;
      return Count rem 2 = 1;
   end Is_Good;

   function Benchmark(N: Integer) return Integer is
      ([for J in 1 .. N =>
          (if Is_Good(J) then 1 else 0)]'Reduce("+",0));

begin
   Ada.Integer_Text_IO.Put(Benchmark(25000));
end Ada_Benchmark;
```

Команда използвана за компилация:

```
gnatmake -O3 -gnat2022 ada_benchmark.adb
```

Забележка. Ада 2022 позволява да направим следното леко изменение на функцията `Benchmark`:

```
function Benchmark(N: Integer) return Integer is
   ([parallel for J in 1 .. N =>
       (if Is_Good(J) then 1 else 0)]'Reduce("+",0));
```

По този начин програмата ще се паралелизира. Ускорението, което може да очакваме, е подобно на ускорението, което се получава при програмата на ръст с използване на щайгата `Rayon`. За съжаление по времето, когато се правеха тези тестове тази възможност на езика ада все още не бе реализирана в компилатора GNAT.

Програма на фортран:

```
      FUNCTION ISPRIME(N)
      ISPRIME = 1
      DO 10 J = 2, N/2
         IF (MOD(N, J) .EQ. 0) THEN
            ISPRIME = 0
            EXIT
         END IF
 10   CONTINUE
      END FUNCTION ISPRIME

      FUNCTION ISGOOD(N)
      K = 0
      DO 20 J = 2, N/2
         IF (MOD(N, J) .EQ. 0) THEN
```

```
            IF (ISPRIME(J) .EQ. 1) THEN
               K = K + 1
            END IF
         END IF
 20   CONTINUE
      ISGOOD = MOD(K, 2)
      END FUNCTION ISGOOD

      PROGRAM BENCHMARK
      K = 0
      DO 30 J = 1, 25000
 30      K = K + ISGOOD(J)
      PRINT *, "Result: ", K
      END
```

Команда използвана за компилация:

```
gfortran -O3 fortran.f --std=legacy -o fortran
```

Програма на си:

```
#include <stdio.h>
#include <stdbool.h>

bool is_prime(int n) {
  int k;
  for(k=2; k<=n/2; k++)
    if (n % k == 0)
      return false;
  return true;
}

int main(void) {
  int count, n, k, i;

  count = 0;
  for(n=2; n<=25000; n++){
    k = 0;
    for(i=2; i<=n/2; i++)
      if (n % i == 0 && is_prime(i))
        k++;
    if (k % 2 != 0)
      count++;
  }

  printf("Result: %i\n", count);
}
```

Команда използвана за компилация с GCC:

```
gcc -O3 c.c -o c-gcc
```

Команда използвана за компилация с Clang:

```
clang -O3 c.c -o c-clang
```

Програма на чепъл:

```
proc is_prime(n: int)
{
   return &&reduce for i in 2..n/2 do n%i != 0;
}

proc prime_divisors(n: int)
{
   return +reduce for i in 2..n/2 do (n%i == 0 && is_prime(i));
}

proc benchmark(n: int)
{
   return +reduce forall i in 1..n do prime_divisors(i) % 2;
}

writeln(benchmark(25000));
```

Команда използвана за компилация:

```
chpl --fast chapel.chpl
```

Програма на ръст:

```
fn is_prime(n: u64) -> bool {
    (2..=n/2).all(|i| n % i != 0)
}

fn is_good(n: u64) -> bool {
    (2..=n/2).filter(|&i| n % i == 0 && is_prime(i)).count() % 2 == 1
}

fn benchmark(n: u64) -> u64 {
    (1..=n).filter(|&i| is_good(i)).count() as u64
}

fn main() {
    println!("Result: {}", benchmark(25000));
}
```

Команда използвана за компилация:

```
rustc -C opt-level=3 rust.rs
```

Забележка. Ако използваме щайгата `Rayon` и изменим дефиницията на функцията `benchmark` по следния начин:

```
use rayon::prelude::*;
fn benchmark(n: u64) -> u64 {
    (1..=n).into_par_iter().filter(|&i| is_good(i)).count() as u64
}
```

тогава ръст автоматично ще паралелизира програмата, така че бързината ѝ ще зависи много от това колко ядра има процесорът. Процесорът на машината, на която се извършваха тестовете, има четири ядра. На него паралелизираната по този начин програма на ръст се изпълнява за 0,1843 милисекунди, което е около 3,7 пъти по-бързо, отколкото бързодействието на последователната програма.

Програма на джава:

```
public class Java {
    static boolean is_prime(int n) {
        for(int k=2; k<=n/2; k++) {
            if (n % k == 0)
                return false;
        }
        return true;
    }
    public static void main(String[] args) {
        int count, k;
        count = 0;
        for(int n=2; n<=25000; n++){
            k = 0;
            for(int i=2; i<=n/2; i++)
                if (n % i == 0 && is_prime(i))
                    k++;
            if (k % 2 != 0)
                count++;
        }
        System.out.println("Result: " + count);
    }
}
```

Команди използвани за компилация и стартиране:

```
javac Java.java
java Java
```

Програма на си# (автор Яна Георгиева):

```
using System;

public class Program {
    public static bool isPrime(int n) {
        int k;
        for (k = 2; k <= n / 2; k++) {
            if (n % k == 0)
                return false;
        }
        return true;
    }


    public static void Main() {
        int count, n, k, i;
        count = 0;
        for (n = 2; n <= 25000; n++) {
            k = 0;
            for (i = 2; i <= n / 2; i++)
                if (n % i == 0 && Program.isPrime(i))
                    k++;
            if (k % 2 != 0)
                count++;
        }
        Console.WriteLine("Result: " + count);
    }
}
```

Команда използвана за компилация:

```
mcs -optimize -out:cSharp.exe cSharp.cs
```

Програма на суифт (автор Яна Георгиева):

```
import Foundation

func isPrime(_ number: Int) -> Bool {
    if number == 2 || number == 3 { return true }
    let maxDivider = Int(number / 2)
    return number > 1 && !(2...maxDivider).contains { number % $0 == 0 }
}

func goodNumbers(_ number: Int) -> Int {
    let maxDivider = Int(number / 2)
    if maxDivider < 2 { return 0 }
    return (2...maxDivider).filter { number % $0 == 0 && isPrime($0) }.count
}

let count = (1...25000).filter { goodNumbers($0) % 2 == 1 }.count
print("Result: \(count)\n")
```

Команда използвана за компилация:

```
swiftc -O swift.swift -o swift
```

Команда използвана за интерпретация:

```
swift swift.swift
```

Програма на джаваскрипт (автор Борислав Ризов):

```
const is_prime = n => {
    const m = Math.floor(n/2) + 1;
    for (let j = 2; j < m; ++j)
        if (!(n%j)) return false;
    return true;
}

const prime_divisors = (n) => {
    let s = 0;
    const m = Math.floor(n/2) + 1;
    for (let j = 2; j < m; ++j) {
        s += (n%j == 0 && is_prime(j));
    }
    return s;
}

const main = () => {
    let benchmark = 0;
    for (let j = 2; j <= 25000; ++j) {
        benchmark += (prime_divisors(j) % 2) == 1;
    }
    console.log(benchmark);
}

main();
```

Програма на го (автор Яна Георгиева):

```
package main

import "fmt"

func isPrime(value int) bool {
    for i := 2; i <= int(value/2); i++ {
        if value%i == 0 {
            return false
        }
    }
    return value > 1
}

func main() {
    cnt := 0
    for i := 2; i <= 25000; i++ {
        k := 0
        for j := 2; j <= i/2; j++ {
            if i%j == 0 && isPrime(j) {
                k++
            }
        }
        if k%2 == 1 {
            cnt++
        }
    }
    fmt.Println(cnt)
}
```

Команда използвана за компилация:

```
go build -o go go.go
```

Команда използвана за компилация с GCC:

```
gccgo -O3 go.go -o go-gcc
```

Програма на паскал:

```
Program Benchmark(Output);

Function Is_prime(N : integer) : boolean;

   Label
      FUNCTION_END;

   Var
      J : integer;

   begin
   Is_prime := True;
   For J := 2 to N div 2 do
      If N mod J = 0 then
         begin
         Is_prime := False;
         Goto FUNCTION_END;
         end;
   FUNCTION_END:
```

```
   end;

Function Is_good(N : integer) : boolean;

   Var
      J, Count : integer;

   begin
   Count := 0;
   For J := 2 to N div 2 do
      If N mod J = 0 then
         If Is_prime(J) then
            Count := Count + 1;
   Is_good := Count mod 2 = 1
   end;

Function Benchmark(N : integer) : integer;

   Var
      J : integer;

   begin
   Benchmark := 0;
   For J := 1 to N do
      If Is_good(J) then
         Benchmark := Benchmark + 1
   end;

begin
Writeln(Benchmark(25000))
end.
```

Команда използвана за компилация:

```
fpc -O3 pascal.pas
```

Програма на форт:

```
: ?PRIME
   DUP 2/ 1+  2 ?DO
      DUP I MOD 0= IF  DROP 0 LEAVE  THEN
   LOOP ;
: #PRIME-DIVISORS
   0  OVER 2/ 1+ 2 ?DO
      OVER I MOD 0= IF  I ?PRIME IF 1+ THEN THEN
   LOOP SWAP DROP ;
: ?GOOD-NUMBER    #PRIME-DIVISORS 2 MOD ;
: #GOOD-NUMBERS   0 SWAP 1+ 2 ?DO I ?GOOD-NUMBER IF 1+ THEN LOOP ;
: BENCHMARK    25000 #GOOD-NUMBERS . ;
```

Команда използвана за компилация:

```
gforth-fast forth.forth -e benchmark -e bye
```

Програма на мъркюри:

```
:- module mercury_benchmark.
:- interface.
:- import_module io.

:- pred main(io::di, io::uo) is det.

:- implementation.
:- import_module int.

:- pred has_divisor(int::in, int::in, int::in) is semidet.

has_divisor(N, From, To) :-
    From =< To,
    (  0 = N mod From
    ;  has_divisor(N, From + 1, To)  ).

:- pred prime_number(int::in) is semidet.

prime_number(N) :-
    not(has_divisor(N, 2, N div 2)).

:- pred prime_divisor(int::in, int::in) is semidet.

prime_divisor(N, K) :- 0 = N mod K, prime_number(K).

:- pred number_of_prime_divisors(int::in, int::in, int::in, int::out) is det.

number_of_prime_divisors(N, From, To, K) :-
    (   From > To
    ->  K = 0
    ;   number_of_prime_divisors(N, From + 1, To, K1),
        (   prime_divisor(N, From)
        ->  K = K1 + 1
        ;   K = K1  ) ).

:- pred number_of_good_numbers(int::in, int::in, int::out) is det.

number_of_good_numbers(From, To, N) :-
    (   From > To
    ->  N = 0
    ;   number_of_good_numbers(From + 1, To, N1),
        (   number_of_prime_divisors(From, 2, From // 2, K),
            1 = K mod 2
        ->  N = N1 + 1
        ;   N = N1  ) ).

main(!IO) :-
    number_of_good_numbers(1, 25000, N),
    io.write_int(N, !IO).
```

Програма на пайтън:

```
#!/usr/bin/python

def is_prime(n):
    return all(n%j for j in range(2, 1+n//2))

def is_good(n):
    return sum(n%j == 0 and is_prime(j) for j in range(2, 1+n//2)) % 2 == 1

benchmark = sum(is_good(j) for j in range(1,25001))

print(benchmark)
```

Програма на окамъл:

```
let prime_number n =
    let rec has_divisor d =
      2 * d <= n && (n mod d = 0 || has_divisor (d+1)) in
    not(has_divisor 2);;

let rec sum_tests test first last =
    if first > last then 0 else
       (if (test first) then 1 else 0)
            + (sum_tests test (first+1) last);;

let is_prime_divisor n k = 0 == n mod k && prime_number k;;

let prime_divisors n = sum_tests (is_prime_divisor n) 2 (n/2);;

let is_good_number n = 1 == (prime_divisors n) mod 2;;

let benchmark n = sum_tests is_good_number 1 n;;
```

Програма на скийм:

```
(define (prime-number n)
  (letrec ([has-divisor (lambda (d)
                          (and (<= (* 2 d) n)
                               (or (= (remainder n d) 0)
                                   (has-divisor (+ d 1)))))])
    (not (has-divisor 2))))

(define (sum-tests test first last)
  (if (> first last) 0
      (+ (if (test first) 1 0)
         (sum-tests test (+ first 1) last))))

(define (is-prime-divisor n)
  (lambda (k)
    (and (= (remainder n k) 0) (prime-number k))))

(define (prime-divisors n)
    (sum-tests (is-prime-divisor n) 2 (/ n 2)))

(define (is-good-number n)
```

```
  (= (remainder (prime-divisors n) 2) 1))

(define (benchmark n)
  (sum-tests is-good-number 1 n))

(write (benchmark 25000))
```

Команда използвана за компилация с Chez:

```
chezscheme --optimize-level 3 --script scheme.scm
```

Команда използвана за компилация с Chicken Scheme:

```
csc scheme.scm
```

Команда използвана за изпълнение с Guile:

```
guile scheme.scm
```

Програма на пролог:

```
divisor(N, K) :-
    N2 is N // 2,
    between(2, N2, K),
    0 is N mod K.

prime_number(N) :- \+ divisor(N,_).

prime_divisor(N, K) :- divisor(N, K), prime_number(K).

odd_number_of_prime_divisors(N) :-
    findall(K, prime_divisor(N, K), X),
    length(X, M),
    1 is M mod 2.

benchmark(N) :-
    findall(K, (  between(1, 25000, K),
                  odd_number_of_prime_divisors(K)  ),
            X),
    length(X, N).
```

Програма на строубери пролог:

```
divisor(K, N) :-
    N2 is N // 2,
    for(K, 2, N2),
    0 is N mod K.

prime_number(N) :- not(divisor(_,N)).

prime_divisor(K, N) :- divisor(K, N), prime_number(K).

odd_number_of_prime_divisors(N) :-
    M is count_successes(prime_divisor(K, N)),
    1 is M mod 2.

benchmark(N) :-
    N is count_successes(call(  for(K, 1, 25000),
                                odd_number_of_prime_divisors(K)  )
                         ).
```

Програма на пиейчпи (автор Яна Георгиева):

```
<?php

function isPrime($n)
{
    for ($k = 2; $k <= $n / 2; $k++) {
        if ($n % $k == 0)
            return false;
    }
    return true;
}


$count = 0;
for ($n = 1; $n <= 25000; $n++) {
    $k = 0;
    for ($i = 2; $i <= $n / 2; $i++)
        if ($n % $i == 0 && isPrime($i))
            $k++;
    if ($k % 2 != 0)
        $count++;
}

echo "Result: " . $count ."\n";
```

Програма на пърл:

```
#!/usr/bin/perl

use strict;
use integer;

sub is_prime {
    my $n = $_[0];
    for my $k (2 .. $n/2) {
        return 0 if ($n % $k == 0);
    }
   return 1;
}

my $count = 0;

for my $n (2 .. 25000) {
    my $k = 0;
    for my $i (2 .. $n/2) {
        $k++ if ($n % $i == 0 && is_prime($i));
    }
    $count++ if ($k % 2 != 0);
}
printf "%i\n",  $count;
```

Програма на хаскел:

```
divisors n = [x | x <- [2..(div n 2)], mod n x == 0]

isPrime n = null (divisors n)

primeDivisors n =  [x | x <- divisors n, isPrime x]

goodNumber n = mod (length (primeDivisors n)) 2 == 1

benchmark = length [x | x <- [1..25000], goodNumber x]

main = print benchmark
```

Програма на ар:

```
Prime <- Vectorize(
    function(num) {
        k <- floor(num/2)
        seq <- seq(2,k,length = max(0,k-2+1))
        ! any(num %% seq == 0)
    }
)

isGoodNumber <- Vectorize(
    function(num) {
        k <- floor(num/2)
        seq <- seq(2,k,length = max(0,k-2+1))
        sum(unlist(Prime(seq[num %% seq == 0]))) %% 2 == 1
    }
)

benchmark <- function(num) { sum(isGoodNumber(1:num)) }
```

Програма на руби:

```
#!/usr/bin/ruby

def is_prime?(n)
  (2..n/2).none? { |i| (n % i).zero? }
end

def is_good?(n)
  (2..n/2).count { |i| (n % i).zero? and is_prime?(i) }.odd?
end

def benchmark(n)
  (1..n).count { |i| is_good?(i) }
end

puts benchmark(25000)
```

Програма на бейсик:

```
10 LET COUNT = 0
20 FOR N = 3 TO 25000
30 LET K = 0
40 FOR I = 2 TO INT(N/2)
50 IF N MOD I <> 0 THEN GOTO 120
60 LET M = 2
70 IF M > INT(I/2) THEN GOTO 110
80 IF I MOD M = 0 THEN GOTO 120
90 LET M = M + 1
100 GOTO 70
110 LET K = K + 1
120 NEXT I
130 IF K MOD 2 <> 0 THEN LET COUNT = COUNT + 1
140 NEXT N
150 PRINT "Result: "; COUNT
160 END
```

Библиография

Нека се занимаем сега със списъка на автори, какъвто има в други книги и какъвто липсва във вашата. Лек за това се намира много лесно, ще трябва само да намерите книга, която да съдържа пълен списък с автори от първата до последната буква на азбуката, както сам вие казахте. Същия този азбучен списък поставете във вашата книга и не се тревожете никак дори ако се открие измамата, поради това, че вие не сте имали никаква нужда да черпите знания от тези автори. Все ще се намери някой наивен читател, който да повярва, че сте използвали всички тези автори за вашата проста и обикновена история. Накрая, ако този дълъг поменик от автори не послужи за друго, то поне ще даде тежест на вашата книга. Още повече че никой не ще седне да проверява дали наистина сте чели, или не тези автори, защото от това няма да му стане нито по-топло, нито по-студено.

Мигел де Сервантес Сааведра *[25]*

[1] Jesse B. Wright Arthur W. Burks, Don W. Warren. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8(46):53–57, 1954.

[2] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK, 2012.

[3] J.A. Bergstra and J.V. Tucker. A characterisation of computable data types by means of a finite, equational specification method. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland,*

July 14-18, 1980, Proceedings, volume 85 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 1980.

[4] J.A. Bergstra and J.V. Tucker. The completeness of the algebraic specification methods for computable data types. *Information and Control*, 54(3):186 – 200, 1982.

[5] Garrett Birkhoff. On the structure of abstract algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31:433–454, 10 1935.

[6] Susanne Bobzien. Ancient logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Center for the Study of Language and Information (CSLI), Stanford University, spring 2016 edition, 2016.

[7] Chin-Liang Chang and Richard C. T. Lee. *Symbolic logic and mechanical theorem proving*. Computer science classics. Academic Press, 1973.

[8] H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

[9] L. L. Ivanov. *Algebraic recursion theory*. Ellis Horwood series in mathematics and its applications: Statistics and operational research. E. Horwood, 1986.

[10] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[11] Michael G. Main and David B. Benson. Free semiring-representations and nondeterminism. *Journal of Computer and System Sciences*, 30(3):318 – 328, 1985.

[12] Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

[13] I. McGilchrist. *The Master and His Emissary: The Divided Brain and the Making of the Western World*. Yale University Press, 2009.

[14] Dauda Odunayo Olanloye. An expert system for diagnosing faults in motorcycle. *International Journal of Engineering and Applied Sciences*, 5(06), 2014.

[15] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[16] Dimiter Skordev. *Computability in combinatory spaces. An algebraic generalization of abstract first order computability*. Kluwer Academic Publishers, Dordrecht-Boston-London, 1992.

[17] Ivan Soskov. On the computational power of the logic programs. In P. Petkov, editor, *Heyting'88 (Varna)*, pages 117–137. Plenum Press, 1990.

[18] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1986.

[19] Robert D. Tennent. *Semantics of programming languages*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[20] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[21] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.

[22] Eliezer Yudkowsky. Cognitive biases potentially affecting judgment of global risks. In Nick Bostrom and Milan Cirkovic, editors, *Global Catastrophic Risks*. Oxford University Press, August 2006.

[23] Анатолий Иванович Мальцев. Несколько замечаний о квазимногообразиях алгебраических систем. *Алгебра и логика (семинар) 5*, 3:3–9, 1966.

[24] Анатолий Иванович Мальцев. *Алгебраические системы*. Современная алгебра. Наука, 1970.

[25] Мигел де Сервантес Сааведра. *Знаменитият идалго Дон Кихот де ла Манча*. Библиотека за ученика. Държавно издателство „Отечество“, София, 1986. Четвърто съкратено издание. Превод на Тодор Нейков.

[26] Димитър Скордев. Логическо програмиране. Записки. `http://http://www.fmi.uni-sofia.bg/fmi/logic/skordev/ln/lp/new/sydyrzha.htm`.

[27] Димитър Скордев. *Комбинаторные пространства и рекурсивность в них*. Изд. на БАН, 1980.

[28] Иван Сосков. Пример полного по Московакису базиса, которы не является программного полным. In А. П. Ершов, editor, *Математическая теория и практика систем программного обеспечения (Новосибирск)*. ВЦ СОАН, 1982.

Изчислителни машини и програмни езици

Организации и фирми

Именен указател

Предметен указател